Distributed Protocols with Threshold and General Trust Assumptions

Inaugural dissertation
of the Faculty of Science,
University of Bern

presented by

Orestis Alpos

from Greece

Supervisor of the doctoral thesis:

Prof. Dr. Christian Cachin
University of Bern, Switzerland

Distributed Protocols with Threshold and General Trust Assumptions

Inaugural dissertation
of the Faculty of Science,
University of Bern

presented by

Orestis Alpos

from Greece

Supervisor of the doctoral thesis:

Prof. Dr. Christian Cachin
University of Bern, Switzerland

Accepted by the Faculty of Science.

Bern,
23rd August 2023

The Dean
Prof. Dr. Marco Herwegh

Acknowledgments

I would, first of all, like to thank Christian, my advisor. His passion and commitment to research, positive attitude, and unwavering optimism have made the past four years truly unforgettable. His continuous support and mentorship have shaped me into a better colleague and a more accomplished scientist. Furthermore, his personal care and dedication to all his students have often made the office feel to me like a second family. With all my heart, thank you, Christian!

I am also grateful to all my colleagues and co-authors who have been an integral part of my journey. The invaluable moments we have shared together and all our discussions have enriched my understanding and personal growth.

I would also like to extend my deepest appreciation to my mother for her relentless efforts, sometimes even sacrifices, which have enabled me to surpass my goals in life. Similarly, I express my gratitude to my father for his unwavering belief in me and boundless love, which have always provided me with strength. I am also fortunate to have two amazing siblings, Mirsini and Thomas, who have been an inseparable part in all my happy childhood memories and family moments.

To Estelle, my girlfriend, I am grateful for her absurd understanding and unparalleled support. Her presence in my life has been an immense source of happiness and strength.

I cannot overlook the impact of my dearest friends from Agrinio, Athens, and Bern. The memories we have created together, the shared time, adventurous trips, our conversations over beers, the unforgettable concerts, the countless jokes and laughter, they all hold an extraordinary place in my heart. Particularly thankful I am to Vasiliki, with whom I have spent unforgettable moments, and who has inspired me to embark on a journey of self-discovery and become the person I am today.

I feel, however, I would be negligent not to acknowledge the most

important person in this journey, the person that has tried its hardest to understand me, that has been next to me in all my winters and found the strength to remain there until the spring, the only person that truly comprehends the sacrifices and rewards of my journey. To my future self, as you read these lines and reflect upon this thesis, I implore you to remember the enormity of your accomplishments and realize that your true power resides within.

> *"I wish it need not have happened in my time," said Frodo. "So do I," said Gandalf, "and so do all who live to see such times. But that is not for them to decide. All we have to decide is what to do with the time that is given us."[156]*

Abstract

Distributed systems today power almost all online applications. Consequently, a wide range of distributed protocols, such as consensus, and distributed cryptographic primitives are being researched and deployed in practice. This thesis addresses multiple aspects of distributed protocols and cryptographic schemes, enhancing their resilience, efficiency, and scalability.

Fundamental to every secure distributed protocols are its trust assumptions. These assumptions not only measure a protocol's resilience but also determine its scope of application, as well as, in some sense, the expressiveness and freedom of the participating parties. Dominant in practice is so far the *threshold* setting, where at most some f out of the n parties may fail in any execution. However, in this setting, all parties are viewed as identical, making correlations indescribable. These constraints can be surpassed with *general* trust assumptions, which allow arbitrary sets of parties to fail in an execution. Despite significant theoretical efforts, relevant practical aspects of this setting are yet to be addressed. Our work fills this gap. We show how general trust assumptions can be efficiently specified, encoded, and used in distributed protocols and cryptographic schemes. Additionally, we investigate a consensus protocol and distributed cryptographic schemes with general trust assumptions. Moreover, we show how the general trust assumptions of different systems, with intersecting or disjoint sets of participants, can be composed into a unified system.

When it comes to decentralized systems, such as blockchains, efficiency and scalability are often compromised due to the total ordering of all user transactions. Guerraoui *et al.* (Distributed Computing, 2022) have contradicted the common design of major blockchains, proving that consensus is not required to prevent double-spending in a cryptocurrency. Modern blockchains support a variety of distributed applications

beyond cryptocurrencies, which let users execute arbitrary code in a distributed and decentralized fashion. In this work we explore the synchronization requirements of a family of Ethereum smart contracts and formally establish the subsets of participants that need to synchronize their transactions.

Moreover, a common requirement of all asynchronous consensus protocols is randomness. A simple and efficient approach is to employ threshold cryptography for this. However, this necessitates in practice a distributed setup protocol, often leading to performance bottlenecks. Blum *et al.* (TCC 2020) propose a solution bypassing this requirement, which is, however, practically inefficient, due to the employment of fully homomorphic encryption. Recognizing that randomness for consensus does not need to be perfect (that is, always unpredictable and agreed-upon) we propose a practical and concretely-efficient protocol for randomness generation.

Lastly, this thesis addresses the issue of deniability in distributed systems. The problem arises from the fact that a digital signature authenticates a message for an indefinite period. We introduce a scheme that allows the recipients to verify signatures, while allowing plausible deniability for signers. This scheme transforms a polynomial commitment scheme into a digital signature scheme.

Contents

1 Introduction **1**
- 1.1 General trust assumptions . . . 1
- 1.2 Synchronization requirements for efficiency in blockchains 4
- 1.3 Distributed randomness generation . . . 5
- 1.4 Deniability in digital signatures . . . 6
- 1.5 Contributions and organization . . . 7

2 Preliminaries **11**
- 2.1 Byzantine quorum systems . . . 11
- 2.2 General distributed cryptography . . . 13

3 General Byzantine quorum systems and consensus **17**
- 3.1 Introduction . . . 17
- 3.2 Techniques . . . 19
  - 3.2.1 General Byzantine quorum systems as formulas . . 20
  - 3.2.2 General Byzantine quorum systems as monotone span programs . . . 23
  - 3.2.3 Checking for quorums . . . 25
  - 3.2.4 Concrete constructions of general Byzantine quorum systems . . . 27
- 3.3 Consensus using general Byzantine quorums systems . . . 29
- 3.4 Implemented HotStuff . . . 34
- 3.5 Evaluation . . . 34
- 3.6 Discussion . . . 42

4 Composition of general Byzantine quorum systems **43**
- 4.1 Introduction . . . 43
- 4.2 Related work . . . 45

4.3 Defining composition for general Byzantine quorum systems 46
4.4 Composition rules for general Byzantine quorum systems 48
4.5 Discussion 52

5 Synchronization power of token smart contracts 55
5.1 Introduction 56
5.2 Related work 58
5.3 Background 59
5.3.1 Shared memory objects 59
5.3.2 Synchronization power 62
5.4 Defining ERC20 tokens as shared objects 63
5.5 Consensus number of ERC20 tokens 67
5.5.1 Overview of the results 67
5.5.2 Technical results and proofs 69
5.6 Extension to other token standards 78
5.7 Discussion 79

6 Practical distributed cryptography with general trust 81
6.1 Introduction 81
6.2 Related work 84
6.3 Background 85
6.4 Specifying and encoding general access structures 86
6.5 Interpolation on general access structures 88
6.6 Verifiable secret sharing 90
6.7 Common coin 94
6.8 Distributed signatures 98
6.9 Evaluation 100
6.9.1 Benchmarking basic properties of the MSP 101
6.9.2 Running time of verifiable secret sharing 103
6.9.3 Running time of common coin 105
6.9.4 Running time of distributed signatures 106
6.10 Discussion 108

7 Practial large-scale distributed randomness generation 109
7.1 Introduction 109
7.2 Related work 113
7.3 Model 115
7.4 Primitives 116
7.4.1 Public-key encryption with full decryption 116
7.4.2 Threshold coin flip 117

7.4.3 Secret sharing . . . 119
7.4.4 Digital Signatures . . . 119
7.4.5 Seed-generation protocol . . . 119
7.5 Weak honest-dealer coin flip . . . 120
7.6 Weak multiple-dealer coin flip . . . 126
7.7 Setting the parameters . . . 131
7.7.1 Sampling a holding committee for wHDCF . . . 131
7.7.2 Sampling a proposer committee for wMDCF . . . 132
7.8 Analysis of communication complexity . . . 133
7.9 Discussion . . . 135

8 DSKE from polynomial commitments 137
8.1 Introduction . . . 137
8.2 Related work . . . 139
8.3 Background . . . 140
8.4 Digital signatures with key extraction (DSKE) . . . 142
8.5 DSKE from polynomial commitment schemes . . . 144
8.6 Discussion . . . 147

9 Conclusion 149

Bibliography 151

Chapter 1

Introduction

1.1 General trust assumptions

Trust assumptions are a fundamental part of secure distributed computing protocols. On one hand, they capture the limits of a protocol's safety properties, thus characterizing the domains in which it may be deployed safely. On the other hand, they also impose limits on the potential of the protocol and, in some sense, the expressiveness and freedom of the parties, thus restricting the domains in which the protocol will be deployed.

Existing Byzantine consensus protocols address only *threshold failures*, where the participating nodes fail independently of each other, each one fails equally likely, and the protocol's guarantees follow from a simple bound on the *number* of faulty nodes. Threshold trust assumptions have been researched and are well understood in practice, but consensus using general trust assumptions has been unexplored.

The same observation can be made for *distributed cryptography*, where independent parties jointly perform some cryptographic task. In the last decade, partly due to blockchains becoming prominent, numerous practical and efficient distributed cryptographic primitives have been deployed. Trust assumptions, however, are only stated through numbers of parties, thus reducing this to *threshold cryptography*, where all parties are treated as identical and correlations cannot be described.

Threshold is not enough. Faults and attacks on the nodes in a system often occur in a coordinated way and exhibit substantial dependencies in practice. Using Werner Vogels' words [159]: "Many academics will confess to have made the assumption that failures of component are not correlated. This absolutely unrealistic assumption will come back to haunt you in real life, where failures frequently are correlated."

Distributed systems and blockchains can benefit from general trust assumptions in the following ways.

Increasing resilience and security. First, general trust assumptions have the capacity to increase the resilience of a system, as failures are, in practice, often correlated. Cyberattacks, exploits, zero-day attacks, and so on very seldom affect all parties in an identical way — they often target a specific operating systems or flavor of it, a specific hardware vendor, or a specific software version. Similarly, attackers may compromise specific parties more easily, due to different administrator policies or different levels of cyber and physical security. In another example, blockchain nodes are typically hosted by cloud providers or mining farms, hence failures are correlated there as well. Such failure correlations are known and have been observed; they can be expressed in a system that supports general trust, significantly increasing resilience and security.

Let us now see a concrete example of how such correlations can be captured. Cachin [31] describes a setting where parties are differentiated in two dimensions, based on their location and operating system (OS). In an instantiation with 16, possibly Byzantine, parties, organized in four locations and four OS, the system tolerates the *simultaneous* failure of all parties in one location and all parties with a specific OS. Hence, it encodes specific knowledge and correlation patterns, and can even tolerate executions with up to seven failed parties, something not possible in the threshold setting, where only five out of the 16 may fail. Once general distributed cryptography is deployed, this example can be generalized to any number of parties and dimensions.

Facilitating personal assumptions and Sybil resistance. Some works in the area of distributed systems generalize trust assumptions in yet another dimension: they allow each party to specify its own. The consensus protocol of Stellar [109], implemented in the Stellar blockchain (https://www.stellar.org/), allows each party to specify the access struc-

ture of its choice, which can consist of arbitrary sets and nested thresholds. Similarly, the consensus protocol implemented by Ripple [146] in the XRP ledger (https://ripple.com/) also allows each party to choose who it trusts and communicates with. In both networks, the resulting representation of trust in the system, obtained when the trust assumptions of all parties are considered together, can only be expressed as a generalized structure. Hence, current threshold-cryptographic schemes cannot be integrated or used on top of these networks. For example, a common coin scheme — necessary for achieving consensus in asynchronous networks — would need to support general trust. In addition to that, practical and easy to deploy general distributed cryptographic schemes can function as a catalyst for more applications built on top of these blockchains.

Another feature of both Stellar and Ripple is that they achieve open membership without employing a proof-of-work or proof-of-stake mechanism. That is, they achieve Sybil resistance by allowing a party to selectively trust or ignore other parties. This approach can lead to more efficient, less energy consuming, and arguably more open and inclusive blockchains. As described earlier, however, this results in trust assumptions where parties are not treated as identical. Departing from a threshold mindset towards general access structures is, thus, a prerequisite for wider adoption.

Open questions. The generalization of threshold trust assumptions and threshold cryptography to any linear access structure is known and typically employs monotone span programs (MSP) [97], a linear-algebraic model of computation. General trust assumptions have been thoroughly explored in theory [113, 114], and cryptographic schemes using an MSP have been described [51, 128, 118, 79]. However, no implementations or deployments exist yet. In our point of view, the reasons are the following.

- Essential implementation details are missing, and usability-related questions have never been answered in a real system. How can the trust assumptions, initially only in the mind of an administrator, be encoded in a scheme or protocol? How does the system administrator efficiently do this? Previous general distributed schemes assume the MSP is given to all algorithms, but how is this built from the trust assumptions? Usability is a necessary ingredient for the adoption of a new technological setting, and usability in turn leads to increased

security.

Research question A : *What is an intuitive, user-friendly, and efficient way to specify and encode general trust assumptions in a protocol?*

- Most importantly, implementations and benchmarks do not exist, hence the efficiency of distributed protocols with general trust and of general cryptographic schemes, is not known. What is the concrete efficiency of the MSP? How does a generalized scheme compare to its threshold counterpart? How much efficiency needs to be "sacrificed" in order to support general trust?

  Research question B : *What is the efficiency of consensus with general trust and of general cryptographic schemes?*

- Current definitions of trust are rigid, non-flexible, and non-dynamic. Consider two or more systems, run by different and possibly disjoint sets of participants, with general but different assumptions about faults. How can they work together? A trivial solution would be to restart the protocol under new trust assumptions, but this would require manually agreeing on, specifying, and encoding the trust assumption anew.

  Research question C : *Can we compose trust assumptions in a deterministic way, that does not require interaction or agreement on the new assumptions, but also results in a system that tolerates as many faults as possible?*

1.2 Synchronization requirements for efficiency in blockchains

Since the inception of blockchains, it has been widely accepted that blockchain nodes must execute all transactions in the same order to ensure consistency. Hence, transactions are synchronized using protocols that implement *total-order broadcast* or *consensus*. This common theme seems to suggest that total order is necessary for the consistency of blockchains.

It was only recently recognized that consensus is not necessary to prevent double-spending in a cryptocurrency, contrary to common belief.

As Guerraoui *et al.* [86] show, all functions of Bitcoin can be emulated on top of strictly weaker communication protocols. Moreover, if shared accounts are allowed, then consensus is required only among the owners of each account. This result suggests that current implementations may be sacrificing efficiency and scalability because they synchronize transactions much more tightly than actually needed.

Modern blockchains support a variety of distributed applications beyond cryptocurrencies, including *smart contracts*, which let users execute arbitrary code in a distributed and decentralized fashion. Regardless of their intended application, blockchain platforms implicitly assume consensus for the correct execution of a smart contract, thus requiring that all transactions are totally ordered. Here we ask ourselves the question: Is consensus required for the correct execution of smart contracts, or can the results of Guerraoui *et al.* be extended to arbitrary code execution? Hence, we initiate a study in the synchronization requirements of smart contracts. Specifically, we research the synchronization requirements of Ethereum's ERC20 token contract, one of the most widely adopted smart contacts.

Research question D : *Is consensus required for token smart contracts?*

1.3 Distributed randomness generation

It is well known that asynchronous total-order broadcast (ATOB) cannot be deterministic [76]. The necessary randomness is usually modelled as a *common coin* scheme [137], informally defined as a source random values observable by all participants but unpredictable for the adversary [35].

Cachin, Kursawe, and Shoup [35] present an efficient common-coin protocol from *threshold cryptography*. Assuming a trusted setup, the value of a coin with identifier cid can be obtained by having each party create a partial signature share on cid, asynchronously wait for $n-t$ valid signature shares, where n is the number of parties and $t < n/3$ is the corruption bound, and then combine the signature shares and hash the result to obtain a value in $\{0,1\}$. This results in a *perfect coin*, meaning that it is uniformly distributed and agreed-upon with probability 1, while the asymptotic communication complexity in the number of parties is only $O(n^2)$. This approach has inspired a number of theoretical works and practical implementations. For example, drand [65] implements a

randomness beacon from threshold BLS signatures [24, 27]. This is, in turn, used in the consensus protocol of Filecoin [135].

However, threshold cryptography requires a distributed setup protocol in practice. In a proof-of-stake (PoS) setting with dynamic stake, or in any deployment where it is desired to proactively refresh [34] the threshold setup, the setup protocol would have to be executed repeatedly. *Asynchronous distributed key generation* ADKG protocols [57, 1, 2] have been proposed in the literature, but their communication cost is $\Omega(n^3)$. A novel idea by Blum *et al.* [23] circumvents the requirement for ADKG. They show how an existing setup can be used to run an instance of Byzantine agreement and to regenerate the setup itself. Their protocol, however, resorts to fully homomorphic encryption and is, therefore, practically inefficient.

Considering the aforementioned approaches to randomness generation, the following research question arises.

Research question E : *Can we have a practical, large-scale, concretely efficient, and modular asynchronous common-coin protocol, that requires no trusted setup and directly supports the PoS setting and dynamic participation?*

1.4 Deniability in digital signatures

In general, digital signatures can be used to prove authenticity for as long as the signature scheme is not broken and the private key is kept secret. While this "long-lived" authenticity might be useful in some scenarios, it is inherently undesirable for certain types of sensitive communication, for instance, whistleblowing. A particular concern is that the communication could be leaked in the future, which might lead to potential retaliation and extortion. This calls for a scheme that lets signers prove authenticity for a limited period of time, while allowing them to deny having signed any messages afterwards. We argue that such a scheme could offer a desirable degree of protection to signers through deniability against future leaks, while reducing the incentives for criminals to obtain leaked communications for the sole purpose of blackmailing.

Previous work has recognized this necessity [13, 153, 28] and has proposed various solutions. One solution [28] requires an interactive key agreement protocol between the sender and the recipient to agree on session keys before exchanging messages. Another way to achieve

deniability is to simply require the sender to periodically rotate keys and publish their old private keys [153]. The following research question naturally arises from the limitations of existing attempts:

Research question F : *Is it possible to design a signature scheme that allows the recipients to verify the validity of the signature, while enabling the sender to gain plausible deniability, without requiring the constant publication of old private keys and the rotation of new public keys?*

1.5 Contributions and organization

This thesis is organized in two parts. Chapters 3 – 5 concern protocols for distributed systems and Chapters 6 – 8 concern cryptographic schemes.

- We start the first part of this thesis by focusing on the generalization of trust assumptions. Targeting Question A, we explore how the general trust assumptions can be specified and encoded within a protocol. We show that the combination of a monotone boolean formula (MBF) and a monotone span program (MSP) leads to efficient implementations. We formulate, implement, and benchmark the HotStuff [163] consensus protocol in the general-trust setting. These are presented in Chapter 3.
- To answer Question C, we define *quorum composition* and its desired properties for systems with general trust assumptions, and show appropriate composition rules. The rules are static, require no interaction or agreement among the participants, and result in a system that tolerates as many faults as possible, subject to necessary liveness and safety properties. These are presented in Chapter 4.
- We then study the synchronization requirements of Ethereum's token smart contracts, focusing on the ERC20 contract and then extending our results to other tokens. We answer Question D by showing that the richer set of methods supported by ERC20 tokens, compared to standard cryptocurrencies, results in strictly stronger synchronization requirements. More surprisingly, the synchronization power of ERC20 tokens depends on the object's state and can thus be modified by method invocations. To prove this result, we develop a dedicated framework to express how the object's state affects the needed synchronization level. This result implies that tailored synchronization protocols, that exploit these dynamic requirements, will lead to more scalable blockchain platforms. We give the details in Chapter 5.

- At this point we switch to the second part of the thesis and present results related to cryptographic schemes. We first answer Question B by presenting distributed schemes with general trust assumptions. Making use of the MBF and MSP encodings, we present a verifiable secret sharing, a common coin, and a distributed signature scheme. We implement all schemes, both in their threshold and general versions, and run benchmarks on multiple general trust assumptions, and conclude that general trust can be used with no significant efficiency loss. We show the results in Chapter 6.
- Targeting Question E, we introduce a simple, modular, and practical common-coin protocol, which can in turn be used in asynchronous total-order broadcast protocols. It is secure in a proof-of-stake setting with *dynamically changing* stake. We adopt the approach of Cachin, Kursawe, and Shoup [35] but remove the requirement for a trusted dealer and apply it to the proof-of-stake setting with dynamic stake. The protocol is shown in Chapter 7.
- Finally, targeting Question F, we introduce the concept of *digital signatures with key extraction (DSKE)*. In a DSKE scheme, the secret key can be extracted if more than a threshold of signatures on arbitrary messages are ever created. Hence, it provides signers with plausible deniability, by demonstrating a group of recipients that can collectively extract the private key, while, within the threshold, each signature still proves authenticity. We give a formal definition of DSKE and a construction from polynomial commitments. We show that, in applications where a signer is expected to create a number of signatures, DSKE offers deniability for free. Moreover, DSKE can be employed to disincentivize malicious behavior, such as equivocation and double-signing. We give the details in Chapter 8.

Publications. The work presented in this thesis is based on the following papers:

- Orestis Alpos, Christian Cachin:
  Consensus Beyond Thresholds: Generalized Byzantine Quorums Made Live.
  International Symposium on Reliable Distributed Systems (SRDS) 2020
- Orestis Alpos, Christian Cachin, Luca Zanolini:
  How to Trust Strangers: Composition of Byzantine Quorum Systems.

International Symposium on Reliable Distributed Systems (SRDS) 2021

- Orestis Alpos, Christian Cachin, Giorgia Azzurra Marson, Luca Zanolini:
  On the Synchronization Power of Token Smart Contracts.
  International Conference on Distributed Computing Systems (ICDCS) 2021
- Orestis Alpos, Zhipeng Wang, Alireza Kavousi, Sze Yiu Chau, Duc Le, Christian Cachin:
  DSKE: Digital Signature with Key Extraction.
  Under submission, 2022.
- Orestis Alpos, Christian Cachin:
  Do Not Trust in Numbers: Practical Distributed Cryptography With General Trust.
  Stabilization, Safety, and Security of Distributed Systems (SSS) 2023
- Orestis Alpos, Christian Cachin, Simon Holmgaard Kamp, Jesper Buus Nielsen:
  Practical Large-Scale Proof-of-Stake Asynchronous Total-Order Broadcast.
  Advances in Financial Technologies (AFT) 2023

Chapter 2

Preliminaries

System model. We refer to the participants in a distributed protocol as *parties* or *processes*, and denote them as $\mathcal{P} = \{p_1, \ldots, p_n\}$. Parties that follow the protocol during an execution are called *honest*, while parties that deviate from its specifications are called *corrupted*. We consider *Byzantine faults*, meaning that corrupted parties are allowed to crash or take arbitrary steps, cooperate, and learn the internal state held by any of them.

Notation. We denote by 1^λ the security parameter and by $\mathsf{negl}(\lambda)$ a negligible function of λ. We denote by $[n]$ the set $\{1, \ldots, n\}$. A bold symbol $\boldsymbol{a}$ denotes a vector of some dimension in $\mathbb{N}^+$. However, we avoid distinguishing between $\boldsymbol{a}$ and $\boldsymbol{a}^\intercal$, that is, $\boldsymbol{a}$ denotes both a row and a column vector. Moreover, $\mathcal{K}$ denotes a finite field, and for vectors $\boldsymbol{a} \in \mathcal{K}^{|a|}$ and $\boldsymbol{b} \in \mathcal{K}^{|b|}$, $\boldsymbol{a} \| \boldsymbol{b} \in \mathcal{K}^{|a|+|b|}$ denotes their concatenation, and a_i is short for $\boldsymbol{a}[i]$. The notation $x \xleftarrow{\$} S$ means that x is chosen uniformly at random from set S.

2.1 Byzantine quorum systems

Secure distributed systems rely on trust assumptions. They define the failures that can be tolerated, and name conditions under which the system may operate. In fault-tolerant replicated systems trust has traditionally been expressed through a threshold: the number of tolerated faulty processes. More generally, trust assumptions are defined through

a *fail-prone system*, which is a collection of subsets of processes, such that each of them contains all the processes that may at most fail together during a protocol execution.

Definition 1 (Fail-prone system [91]). A *fail-prone system* (FPS) is a collection of sets of processes $\mathcal{F} \subset 2^{\mathcal{P}}$, none of which contains another, where each $F \in \mathcal{F}$ is called a *fail-prone set*, such that some $F \in \mathcal{F}$ contains all processes that may at most fail together in some execution.

A complementary structure to the fail-prone system is the *Byzantine quorum system* [113], defined as follows.

Definition 2 (Byzantine quorum system [113]). Let $\mathcal{F} \subseteq 2^{\mathcal{P}}$ be a fail-prone system. A *Byzantine quorum system* (BQS) for the FPS $\mathcal{F}$ is a collection of sets of processes $\mathcal{Q} \subseteq 2^{\mathcal{P}}$, none of which is contained in another, where each $Q \in \mathcal{Q}$ is called a *quorum*, such that:

Consistency:

$$\forall Q_1, Q_2 \in \mathcal{Q}, \forall F \in \mathcal{F} : \; Q_1 \cap Q_2 \not\subseteq F.$$

Availability:

$$\forall F \in \mathcal{F} : \; \exists \; Q \in \mathcal{Q} : \; F \cap Q = \emptyset.$$

A link between the two definitions is given by the following results.

Definition 3 (Q^3-condition [91, 113]). Let $\mathcal{F}$ be a fail-prone system. We say that $\mathcal{F}$ satisfies the *Q^3-condition*, abbreviated as $Q^3(\mathcal{F})$, if it holds

$$\forall F_1, F_2, F_3 \in \mathcal{F} : \; \mathcal{P} \not\subseteq F_1 \cup F_2 \cup F_3.$$

Lemma 1 (Quorum system existence [113]). *Let $\mathcal{F}$ be a fail-prone system. A Byzantine quorum system for $\mathcal{F}$ exists if and only if $Q^3(\mathcal{F})$. In particular, if $Q^3(\mathcal{F})$ holds, then $\overline{\mathcal{F}}$, the bijective complement of $\mathcal{F}$, is a Byzantine quorum system called* canonical quorum system *of $\mathcal{F}$.*

Threshold and general systems. When an FPS or a BQS is defined only by cardinality, i.e., it includes all the subsets of $\mathcal{P}$ of a given size, it is called a *threshold FPS* or *threshold BQS*, respectively. In this case, the Q^3-condition is equivalent to the requirement $n > 3f$, and, if we set $n = 3f + 1$, then $\mathcal{F} = \{F \subset \mathcal{P} : |F| = f\}$ and $\mathcal{Q} = \{Q \subset \mathcal{P} : |Q| = 2f + 1\}$. When an FPS or a BQS is allowed to contain arbitrary subsets of $\mathcal{P}$, subject to the Q^3-condition, it is called a *general BQS* or a *general FPS*, respectively.

Remark 1 (Notation for FPS and BQS). Note that both fail-prone systems and quorum systems contain no elements that are subsets of another element. A fail-prone system is *maximal*, i.e., if $F \in \mathcal{F}$, there is no $F' \in \mathcal{F}$ such that $F' \subset F$, and, a quorum system is *minimal*, i.e., if $Q \in \mathcal{Q}$, there is no $Q' \in \mathcal{Q}$ such that $Q' \supset Q$. In this thesis we use the additional notation $\mathcal{F}^+$ and $\mathcal{Q}^+$ for their monotone extensions, respectively. That is, $\mathcal{F}^+$ contains all the fail-prone sets and their subsets, and $\mathcal{Q}^+$ contains all the quorums and their supersets.

2.2 General distributed cryptography

We next give some definitions that are useful for distributed cryptographic schemes.

Definition 4 (Adversary structures [91] and access structures [21]). An *adversary structure* is a collection of sets of processes $\mathcal{F}^+ \subseteq 2^{\mathcal{P}}$, where each $F \in \mathcal{F}^+$ is called an *unauthorized set*, and an *access structure (AS)* is a collection of sets of processes $\mathcal{A}^+ \subseteq 2^{\mathcal{P}}$, where each $A \in \mathcal{A}^+$ is called an *authorized set*. Both are monotone: any subset of an unauthorized set is unauthorized, i.e., if $F \in \mathcal{F}^+$ and $B \subset F$, then $B \in \mathcal{F}^+$, and any superset of an authorized set is authorized, i.e., if $A \in \mathcal{A}^+$ and $C \supset A$, then $C \in \mathcal{A}^+$. As in the most general case [91, 51] we assume *canonical* access structures, that is, $\mathcal{A}^+$ contains the complement of each set in $\mathcal{F}^+$. We say that $\mathcal{F}^+$ is a Q^2 *adversary structure* if no two sets in $\mathcal{F}^+$ cover the whole $\mathcal{P}$.

Remark 2 (Notation for adversary structures and access structures). Note that in the above definition, and generally in the literature [18, 91, 51], adversary structures and access structures are by definition monotone. In this thesis we use the additional notation $\mathcal{F}$ and $\mathcal{A}$ for their non-monotone versions. That is, $\mathcal{F}$ denotes the maximal adversary structure, which contains only the *maximal unauthorized sets*, and $\mathcal{A}$ denotes the minimal access structure, which contains only the *minimal authorized sets*. However, when it is clear from the context, we may not distinguish between $\mathcal{F}^+$ and $\mathcal{F}$, or $\mathcal{A}^+$ and $\mathcal{A}$. In the literature $\mathcal{F}$ has also been called the *basis* of $\mathcal{F}^+$, and $\mathcal{A}$ the *basis* of $\mathcal{A}^+$.

We now define the notion of *insertion* on AS, which allows to recursively create AS by combining existing, smaller ones.

Definition 5 (Insertion on access structures [117, 91]). Let $\mathcal{A}_1^+$ and $\mathcal{A}_2^+$ be two monotone access structures defined on two sets of parties $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, and let $p_z \in \mathcal{P}_1$ such that $p_z \notin \mathcal{P}_2$. The *insertion* of $\mathcal{A}_2^+$ at p_z, written as $\mathcal{A}_1^+(p_z \rightarrow \mathcal{A}_2^+)$, is the monotone access structure $\mathcal{A}_3^+$ defined on the set $\mathcal{P}_3 = (\mathcal{P}_1 \setminus \{p_z\}) \cup \mathcal{P}_2$ that satisfies the following: a set $A \subseteq \mathcal{P}_3$ is authorized in $\mathcal{A}_3^+$ if and only if the set $A \cap \mathcal{P}_1$ is authorized in $\mathcal{A}_1^+$ or the set $A \cap \mathcal{P}_1$ together with p_z is authorized in $\mathcal{A}_1^+$ and p_z is replaced by a set authorized in $\mathcal{A}_2^+$. Formally,

$$A \in \mathcal{A}_3^+ \Leftrightarrow A \cap \mathcal{P}_1 \in \mathcal{A}_1^+ \vee \left((A \cap \mathcal{P}_1) \cup \{p_z\} \in \mathcal{A}_1^+ \wedge A \cap \mathcal{P}_2 \in \mathcal{A}_2^+\right).$$

Monotone span programs [97]. *Monotone span programs* (MSP) have been introduced as a linear-algebraic model of computation. An MSP is a quadruple $\mathcal{M} = (M, \rho, \boldsymbol{e}_1, \mathcal{P})$, where M is an $m \times d$ matrix over a finite field $\mathcal{K}$, ρ is a surjective function $\{1, \ldots, m\} \rightarrow \{p_1, \ldots, p_n\}$ that labels each row of M with a party in $\mathcal{P}$, and $\boldsymbol{e}_1$ is the vector $(1, 0, \ldots, 0) \in \mathcal{K}^d$, called the *target vector*. If $\boldsymbol{r}_i$ is a row of M and $\rho(i) = p_j, p_j \in \mathcal{P}$, we say that party p_j *owns* row $\boldsymbol{r}_i$. There is also a function $\phi : \mathcal{P} \rightarrow 2^{\{1,\ldots,m\}}$, such that $\phi(p_j)$ is the set of rows owned by party p_j. The *size* of the MSP is the number of its rows m.

For any set $A \subseteq \mathcal{P}$ we define M_A to be the $m_A \times d$ matrix obtained from M by keeping only the rows $\boldsymbol{r}_i$ with $\rho(i) \in A$, that is, only the rows owned by parties in A. Let $M_A^\intercal$ denote the transpose of M_A and $Im(M_A^\intercal)$ the span of the rows of M_A. We say that the MSP *accepts* the set A if the rows of M_A span $\boldsymbol{e}_1$, that is, $\boldsymbol{e}_1 \in Im(M_A^\intercal)$. Equivalently, there is a *recombination vector* $\boldsymbol{\lambda}_A$ such that $\boldsymbol{\lambda}_A M_A = \boldsymbol{e}_1$. We say that the MSP *rejects* A otherwise. For any access structure $\mathcal{A}^+$, we say that an MSP *accepts* $\mathcal{A}^+$ if it accepts exactly the sets in $\mathcal{A}^+$.

It has been proven that each MSP accepts exactly one monotone access structure and each monotone access structure can be expressed in terms of an MSP [18, 97]. Hence, an MSP uniquely defines an access structure, which in turn implies an adversary structure, the canonical one.

Definition 6 (Linear secret sharing). A *linear secret-sharing scheme* (LSSS), defined on adversary structure $\mathcal{F}^+$ and access structure $\mathcal{A}^+$, shares a secret $x \in \mathcal{K}$, for finite field $\mathcal{K}$, using a *coefficient vector* $\boldsymbol{r}$, in such a way that every share is a linear combination of x and the entries of $\boldsymbol{r}$. It satisfies two properties. The first is *correctness*, which demands that any authorized set $A \in \mathcal{A}^+$ can reconstruct the secret. It is satisfied

by construction of the MSP, which accepts the access structure $\mathcal{A}^+$. The second is *privacy*, stating any unauthorized set $F \in \mathcal{F}^+$ obtains no information about the secret. This is formalized by the following lemma.

Lemma 2 (Privacy of linear secret-sharing schemes [97]). *Let $\mathcal{M} = (M, \rho, \boldsymbol{e}_1, \mathcal{P})$ be an MSP over finite field $\mathcal{K}$, which accepts the access structure $\mathcal{A}^+$, and F an unauthorized set, i.e. $F \notin \mathcal{A}^+$, with shares $\boldsymbol{x}_F = M_F\boldsymbol{r}$. Then, for every secret $\tilde{x} \in \mathcal{K}$ there exists a coefficient vector $\tilde{\boldsymbol{r}}$ which shares the secret $\tilde{x}$, i.e., $\tilde{r}_1 = \tilde{x}$, and satisfies $\boldsymbol{x}_F = M_F\tilde{\boldsymbol{r}}$.*

Algorithm 1 (Linear secret-sharing scheme). We formalize an LSSS as two algorithms, *Share()* and *Reconstruct()*.

1. *Share(x)*. Choose uniformly at random $d-1$ elements $r_2, \ldots, r_d$ from $\mathcal{K}$ and define the *coefficient vector* $\boldsymbol{r} = (x, r_2, \ldots, r_d)$. Calculate the secret shares $\boldsymbol{x} = (x_1, \ldots, x_m) = M\boldsymbol{r}$. Each x_j, with $j \in [1, m]$, belongs to party $p_i = \rho(j)$. Hence, p_i receives in total m_j shares, where m_j is the number of MSP rows owned by p_i.
2. *Reconstruct($A, \boldsymbol{x}_A$)*. To reconstruct the secret given an authorized set A and the shares $\boldsymbol{x}_A$ of parties in A, find the recombination vector $\boldsymbol{\lambda}_A$ and compute the secret as $\boldsymbol{\lambda}_A\boldsymbol{x}_A$.

Proof of Lemma 2. Let the dimensions of M be $m \times d$, and let the secret shared by $\boldsymbol{r}$ be x, i.e., $r_1 = x$. By definition of an unauthorized set, the rows of M_F do not span $\boldsymbol{e}_1$. That means, $rank(M_F) < rank(\begin{smallmatrix} M_F \\ \boldsymbol{e}_1 \end{smallmatrix})$ and, from linear algebra, we know that $|kernel(M_F)| > |kernel(\begin{smallmatrix} M_F \\ \boldsymbol{e}_1 \end{smallmatrix})|$. This implies the existence of a vector $\boldsymbol{w} \in \mathcal{K}^d$, $\boldsymbol{w} \neq \boldsymbol{0}$, such that $M_F\boldsymbol{w} = \boldsymbol{0}$ (i.e., $\boldsymbol{w} \in kernel(M_F)$), and $w_1 = 1$ (i.e., $\boldsymbol{w} \notin kernel(\begin{smallmatrix} M_F \\ \boldsymbol{e}_1 \end{smallmatrix})$). Define $\tilde{\boldsymbol{r}} = \boldsymbol{r} + (\tilde{x} - x)\boldsymbol{w}$. Notice that $\tilde{r}_1 = \tilde{x}$, so $\tilde{\boldsymbol{r}}$ shares the secret $\tilde{x}$. Moreover, $M_F\tilde{\boldsymbol{r}} = M_F\boldsymbol{r} + (\tilde{x} - x)M_F\boldsymbol{w} = M_F\tilde{\boldsymbol{r}}$. □

It has been shown that linear secret-sharing schemes are equivalent to monotone span programs [18, 97].

Remark 3 (Lower bounds for general secret sharing). Superpolynomial lower bounds are known for MSP [142, 15] and general secret sharing [103]. As the focus of this work is on practical aspects, we assume that the access structure can, in the first place, be efficiently described by a user or system administrator, either as a collection of sets or as a Boolean formula. Arguably, access structures of practical interest fall in this category. Moreover, it is known that MSPs are more

powerful than Boolean formulas and circuits. Babai, Gál, and Wigderson [15] prove the existence of monotone Boolean functions that can be computed by a linear-size MSP but only by exponential-size monotone Boolean formulas. In those cases the MSP can be directly plugged into our schemes with general trust.

Chapter 3

General Byzantine quorum systems and consensus

3.1 Introduction

Byzantine quorum systems (BQS) [113] are the key abstraction for capturing the trust assumptions in distributed protocols where parties may behave maliciously. By definition, BQS support general quorums, that is, a BQS can contain arbitrary subsets of $\mathcal{P}$. General BQS have been intensely explored in the literature [113]. For example, Malkhi *et al.* [114] study their load and availability, Hirt and Maurer [91] use a very related notion for secure multiparty computation, Junqueira *et al.* [95] explore an equivalent formalization in terms of survivor sets, and Warns *et al.* [161] introduce a generalized model that unifies multiple such failure models.

Nevertheless, these works approach general BQS mainly from a theoretical perspective. When considering practical, state-of-the-art distributed protocols with Byzantine faults, especially state-machine replication (SMR) protocols in the blockchain space, one notices that threshold BQS are the only occurring trust structure. To name some examples, Aublin *et al.* [14] present an abstraction of an SMR protocol and build BFT algorithms using threshold Byzantine quorums. Liu *et al.* [36] in-

troduce cross fault-tolerance (XFT), a model that provides guarantees of crash fault-tolerance but tolerates a number of Byzantine faults. Buchman *et al.* [30] present Tendermint, a consensus protocol based on the classical PBFT [43] algorithm, making use of a novel gossip primitive. Finally, Yin *et al.* introduce HotStuff [163], a BFT SMR protocol with linear communication complexity. Threshold BQS have been researched and well understood in practice, but consensus using *general BQS* has been unexplored.

Related work. Karchmer and Wigderson [97] formally define Monotone Span Programs (MSP) as a model for computation. Many constructions have been suggested for creating the MSP of a given access structure. Lewko and Waters [105] suggest a general algorithm for converting any monotone Boolean formula to an MSP, that is however inefficient for access structures expressed with threshold operators. The notion of *insertion* has been introduced by Martin [117]. Nikov and Nikova [127] explore constructions for recursively building the MSP for an access structure from existing MSPs for smaller access structures, and presented the definition of insertion, that we also use in our work.

Contributions. In this chapter we focus on general BQS and demonstrate the first BFT consensus protocol with general trust. We describe all components necessary for general BQS-based protocols and investigate different ways to realize them. In particular, we address all the following topics:

Encoding a general BQS: We first consider a monotone Boolean formula (MBF) consisting of *and*, *or*, and *threshold* operators for specifying a BQS. Since monotone span programs are stronger than monotone Boolean formulas, as mentioned, we also investigate MSP for representing general BQS. We exhibit an algorithm for turning a BQS specification into an MSP. When the BQS is specified as a monotone formula, the size of the created MSP is linear in its inputs.

Integrating general BQS with consensus: For both representations (MBF and MSP), we show algorithms for checking quorum properties and for integrating them with distributed protocols. Comparing the implementations we observe that the MBF-based method generally performs better than the MSP-based implementation because of the matrix manipulations required by the MSP. This provides the

first unified treatment of the efficiencies of these methods and paves the way for their practical deployment.

General Byzantine quorum systems: We apply our methods to general BQS as described in the literature. For the M-Grid BQS [114], which arranges n nodes in a square and tolerates $O(\sqrt{n})$ Byzantine nodes, we construct the corresponding MSP and investigate its properties. We implement an attribute-defined BQS generalizing the OS and location-based example mentioned before and represent this as an MBF and as an MSP.

HotStuff consensus with general BQS: Last but not least, we address consensus, the central problem in distributed computing. Applying our approach, we realize consensus with general BQS by building on HotStuff [163], an efficient BFT consensus algorithm. This is the first BFT consensus implementation using a generalized trust assumption. In benchmarks with up to 40 replicas, we observe that the performance with the MBF representation is comparable to that of the threshold BQS. Using the same threshold trust structure, the MSP representation shows lower performance.

Organization. The rest of this chapter is organized as follows. Section 3.2 presents our techniques for encoding a BQS. In Section 3.3 we describe HotStuff consensus algorithm with general BQS and prove its consistency and liveness properties. Section 3.5 subsequently evaluates an implementation of our general BQS methods using the HotStuff consensus protocol.

System model. As presented in Section 2.1, the corruption capabilities of the adversary are specified by a *fail-prone system* $\mathcal{F}$. For a specific execution we denote as B the set of the *actually faulty parties*.

3.2 Techniques

When Byzantine quorum systems are allowed to contain arbitrary sets, two questions arise: How will these sets be specified by the user? And how are they encoded within a protocol? A first solution could involve an enumeration of all quorums, this would however lead to long user-inputs and large internal representation. A more efficient solution is hence

required, one that provides users with an effective, intuitive and user-friendly way to specify a BQS. It is also crucial to internally encode the BQS using a data structure that is efficient, able to encode any possible BQS, and also offering an inexpensive method for checking whether a set is a quorum.

Remark 4. The following sections describe algorithms to construct a monotone Boolean formula (MBF) and a monotone span program (MSP) for a Byzantine quorum system $\mathcal{Q}$ or for an access structure $\mathcal{A}$, as well as an algorithm to check whether a given set is a quorum or an authorized set. The input to the algorithms can be some description (such as a JSON-based encoding as in Figure 3.2) of either $\mathcal{Q}$ or $\mathcal{Q}^+$, in the case of a quorum system, and of either $\mathcal{A}$ or $\mathcal{A}^+$, in the case of an access structure. We remind the reader that $\mathcal{Q}^+$ contains all quorums in $\mathcal{Q}$ and all their supersets, and similarly of $\mathcal{A}$ and $\mathcal{A}^+$. The resulting MBF and MSP encode $\mathcal{Q}^+$ or $\mathcal{A}^+$, that is, they return 1 on input any quorum or superset of a quorum, in the case of a quorum system, and any authorized set or superset of an authorized set, in the case of an access structure.

3.2.1 General Byzantine quorum systems as formulas

In this section we show how the generalized trust assumptions of the system can be specified by the user in a structured way and encoded within the protocol as a Boolean formula. This technique will be taken up again and extended in Chapter 6.

We observe that it is enough to use only the *threshold operator* $\Theta_k^m(q_1, \ldots, q_m)$, which specifies that any subset of $\{q_1, \ldots, q_m\}$ with cardinality k is a quorum. Each q_i can be a literal, i.e., a party identifier, or a nested threshold operator. The threshold operator is the generalization of logical *conjunction*, that would require all q_i to make a quorum, and logical *disjunction*, that would allow each of them alone to be a quorum – the first can be obtained for $k = m$ and the second for $k = 1$. The threshold operator is thus complete, in the sense that it can describe any possible BQS. Therefore, the users are allowed to specify the generalized trust assumptions in a standard format like JSON, using nested threshold operators. This is aligned with the way users specify their quorum slices in Stellar Blockchain [120] with threshold operators.

We use the notion of a *monotone Boolean formula* (MBF), a formula that consists of *and*, *or*, and *threshold* operators and literals that corre-

spond to parties. An MBF F describes a monotone function $2^{\mathcal{P}} \to \{0,1\}$ in the following way; when F consists only of a literal, then the value of F on input $S \subseteq \mathcal{P}$ is 1 if and only if $F \in S$; when F is the threshold operator $\Theta_k^m(q_1, \ldots, q_m)$, then $F(S)$ is 1 if at least k of the $q_1, \ldots, q_m$ are recursively evaluated to 1 on input S; and accordingly for the other operators. We say that an MBF F *implements* a BQS $\mathcal{Q}^+$ if it returns 1 on input a set $Q \in \mathcal{Q}^+$, and 0 otherwise.

We use a tree data structure to store a BQS described through an MBF, where the internal nodes represent an operator, their children are the operands, and the leaves always represent a party. Clearly, the size of the tree (defined as the number of nodes) is linear in the quorum specification given by the user. We employ Algorithm 1 to evaluate whether a set is considered a quorum in the BQS implemented by a formula F. The runtime is linear in the size of F, given that the set membership operation returns in constant time.

Algorithm 1 Checking whether set A is a quorum in the BQS implemented by formula F.

```
1: eval(F, A)
2:     if F is a literal then
3:         return (F ∈ A)
4:     else
5:         write F = op(F_1, ..., F_m), where op ∈ {∧, ∨, Θ}
6:         for each F_i do
7:             x_i ← eval(F_i, A)
8:         return op(x_1, ..., x_m)
```

A layered BQS. An example that highlights a more complex BQS that cannot be specified in the threshold model is a *2-layered-1-common* BQS (2L1C). This example shows a hierarchical trust structure with a notion of proximity that models a realistic system structured into two levels. To our knowledge, it has not been used in practice so far. Let us consider two disjoint sets of parties, organized in two layers, with k parties $A_0 \ldots A_{k-1}$ on the first and $3k$ parties $B_0 \ldots B_{3k-1}$ on the second. We may assume that the parties in the first layer are more trusted than those in the second layer. A quorum consists of a strict 2/3 majority of the parties in the first layer plus, for each party A_ℓ of these, a 2-out-of-4 threshold from the set $\{B_{3\ell}, \ldots, B_{3(\ell+1)}\}$, where indices are

modulo $3k$. For $k \in \mathbb{N}$, the general formula of the BQS is

$$\Theta^{k}_{\lceil \frac{2k+1}{3} \rceil}\left(\{A_\ell \wedge \Theta^{4}_{2}(\{B_m\})\}\right), \quad \text{for } \ell \in \{0,\ldots,k-1\} \text{ and } m \in \{3\ell,\ldots,3(\ell+1) \bmod 3k\}. \tag{3.1}$$

A 2-layered-1-common BQS for $k = 4$ can be seen in Figure 3.1. In Figure 3.2 we show a configuration file that specifies this BQS – it is actually the file used during the evaluation. It is worth to notice that this BQS, even for $k = 4$, results in a system with 792 quorums, which highlights why a naive, quorum-enumeration solution would be impractical. Notice that by using the fail-prone system that corresponds to a 2L1C BQS in the canonical way, we observe that this BQS satisfies the Q^3-condition because every fail-prone set contains fewer than $k/3$ parties from the first layer. Thus, it is indeed a BQS.

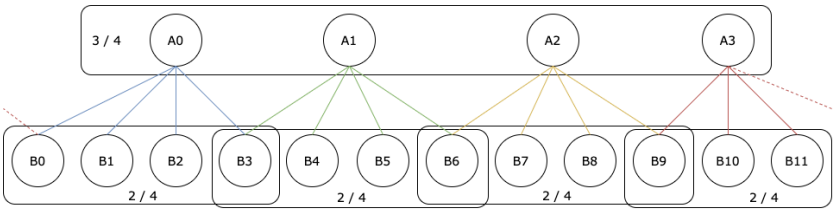


Figure 3.1. The 2L1C Byzantine quorum system for $k = 4$. The corresponding MBF is $\Theta^4_3(A_0 \wedge \Theta^4_2(B_0, B_1, B_2, B_3), A_1 \wedge \Theta^4_2(B_3, B_4, B_5, B_6), A_2 \wedge \Theta^4_2(B_6, B_7, B_8, B_9), A_3 \wedge \Theta^4_2(B_9, B_{10}, B_{11}, B_0))$.

```
{"select": 3, "out-of": [
    {"select": 2, "out-of":["A0", {"select": 2, "out-of":["B0", "B1", "B2", "B3"]} ]},
    {"select": 2, "out-of":["A1", {"select": 2, "out-of":["B3", "B4", "B5", "B6"]} ]},
    {"select": 2, "out-of":["A2", {"select": 2, "out-of":["B6", "B7", "B8", "B9"]} ]},
    {"select": 2, "out-of":["A3", {"select": 2, "out-of":["B9", "B10", "B11", "B1"]} ]}
]}
```

Figure 3.2. A specification of the 2L1C Byzantine quorum system for $k = 4$ in JSON format using nested threshold operators.

3.2.2 General Byzantine quorum systems as monotone span programs

Until now, we considered BQS that can be efficiently encoded using formulas. However, as already discussed in Remark 3, results in complexity theory suggest that MSPs can be superpolynomially stronger than monotone formulas. Moreover, the MSP is a compact and concise data structure, that can be encoded by a matrix and a vector over a field. For these reasons, we also investigate the capabilities of the MSP as the data structure that encodes a BQS. In this section we show how to instantiate an MSP from an MBF and how the MSP can be used to check for quorums. Later, we evaluate the MSP-based implementation and compare it with the one based on MBF. We remark, however, that constructing the MSP from an MBF is not the only option; in case a BQS is more efficiently described by an MSP than by a formula, we could plug the MSP directly in the protocol and use the same quorum-checking algorithms.

In line with our previous terminology, we say that an MSP $\mathcal{M}$ *implements* an access structure $\mathcal{A}$ if it accepts exactly the sets in $\mathcal{A}$ and their supersets. Returning to the idea of *insertion*, we first show how this notion is reflected on the MSP that implements an access structure. In the following, let $\mathcal{M}^{(k)} = (M^{(k)}, \rho^{(k)}, \boldsymbol{e}_1^{(k)}, \mathcal{P}^{(k)})$ be MSPs, where $M^{(k)}$ has dimensions $m_k \times d_k$, for $k \in \{1, 2, 3\}$. We denote the rows of each $M^{(k)}$ as $\boldsymbol{r}_i^{(k)}$, for $1 \leq i \leq m_k$. We also denote the j^{th} column in a row $\boldsymbol{r}$ as $\boldsymbol{r}[j]$, a range of columns j_1 to j_2 as $\boldsymbol{r}[j_1 : j_2]$, a row with ℓ zero elements as $\mathbf{0}^\ell$, and the concatenation of two rows $\boldsymbol{r}$ and $\boldsymbol{r}'$, that is a new vector of size $|\boldsymbol{r}| + |\boldsymbol{r}'|$, whose first elements are $\boldsymbol{r}$ and the last are $\boldsymbol{r}'$, as $\boldsymbol{r} \mid\mid \boldsymbol{r}'$.

Definition 7 (Insertion on MSPs [127]). Let $\boldsymbol{r}_z$ be a row of $M^{(1)}$ owned by $p_z \in \mathcal{P}^{(1)}$ – assuming without loss of generality it is unique. The *insertion* of $M^{(2)}$ in row $\boldsymbol{r}_z$ of $M^{(1)}$, written as $\mathcal{M}^{(1)}(\boldsymbol{r}_z \to \mathcal{M}^{(2)})$, is an MSP $\mathcal{M}^{(3)}$, where $M^{(3)}$ has rows identical to $M^{(1)}$, except for $\boldsymbol{r}_z$, which is repeated m_2 times in $M^{(3)}$, each time multiplied by the first column of $M^{(2)}$ and with the rest of the columns 2 to d_2 of $M^{(2)}$ appended in the end. The function $\rho^{(3)}$ labels the rows of $M^{(3)}$ with the same owners as $\rho^{(1)}$, except for $\boldsymbol{r}_z$. The newly inserted rows are labeled according to $\rho^{(2)}$.

More formally, $M^{(3)}$ is an $(m_1 + m_2 - 1) \times (d_1 + d_2 - 1)$ matrix with

rows

$$\boldsymbol{r}_i^{(3)} = \begin{cases} \boldsymbol{r}_i^{(1)} \mid\mid \mathbf{0}^{d_2-1} & 1 \le i \le z-1 \\ \boldsymbol{r}_z * \boldsymbol{r}_{i-z+1}^{(2)}[1] \mid\mid \boldsymbol{r}_{i-z+1}^{(2)}[2:d_2] & z \le i \le z+m_2-1 \\ \boldsymbol{r}_{i-m_2+1}^{(1)} \mid\mid \mathbf{0}^{d_2-1} & z+m_2 \le i \le m_1+m_2-1 \end{cases} \tag{3.2}$$

and $\rho^{(3)}$ is a surjective function $\{1,\ldots,m_1+m_2-1\} \to (\mathcal{P}^{(1)} \backslash \{p_z\}) \cup \mathcal{P}^{(2)}$ defined as

$$\rho^{(3)}(i) = \begin{cases} \rho^{(1)}(i) & 1 \le i \le z-1 \\ \rho^{(2)}(i-z+1) & z \le i \le z+m_2-1 \\ \rho^{(1)}(i-m_2+1) & z+m_2 \le i \le m_1+m_2-1 \end{cases}$$

Lemma 3. *[127] If an MSP $\mathcal{M}^{(1)}$ implements the access structure $\mathcal{A}^{(1)}$, with row $\boldsymbol{r}_z$ owned by party p_z, and an MSP $\mathcal{M}^{(2)}$ implements the access structure $\mathcal{A}^{(2)}$, then the MSP $\mathcal{M}^{(1)}(\boldsymbol{r}_z \to \mathcal{M}^{(2)})$ implements the access structure $\mathcal{A}^{(1)}(p_z \to \mathcal{A}^{(2)})$ (see Definition 5).*

Lemma 4. *Let* Vandermonde-MSP$(n,t,\mathcal{P})$ *be defined as the MSP $(V(n,t),\rho,\boldsymbol{e}_1,\mathcal{P})$, with $\mathcal{P} = \{p_1,\ldots,p_n\}$, $V(n,t)$ the $n \times t$ Vandermonde matrix over a finite field $\mathcal{K}$,*

$$V(n,t) = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{t-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{t-1} \end{pmatrix},$$

with $x_i \neq x_j \neq 0$, for $1 \le i \le j \le n$, ρ a function that maps row $\boldsymbol{r}_i$ to party p_i, for $i \in \{1,\ldots,n\}$, and $\boldsymbol{e}_1 = (1,0,\ldots,0) \in \mathcal{K}^t$. Then, Vandermonde-MSP$(n,t,\mathcal{P})$ implements the t out-of n threshold access structure $\Theta_t^n(\mathcal{P})$.

Proof. Let $A \subset \mathcal{P}$ and M_A the matrix consisting of the rows of M owned by the members of A. From the results of linear algebra, and because x_i's are pairwise different, we know that the rank of M_A is maximal (that is, equal to t, and thus $Im(M_A^\intercal) = \mathcal{K}^t$) if and only if $|A| \ge t$. Therefore, $\mathcal{M}$ accepts exactly those sets A with $|A| \ge t$. □

Building the MSP that implements a general BQS. Based on the previous lemmata, we now present Algorithm 2 that gets as input an

MBF (that encodes a Byzantine quorum system or an access structure) and outputs an MSP that implements the Byzantine quorum system or the access strucure. The idea is to start with a Vandermonde matrix implementing the first in the hierarchy threshold operator and repeatedly perform insertions of the MSP implementing the nested threshold operators.

Algorithm 2 works as follows. Let $F = \Theta_d^m(F_1, \ldots, F_m)$ be an MBF, where each F_i can be a party or a nested threshold operator. The algorithm first creates the MSP for F (lines 9–21) in the following way: it extracts the values m, d and $F_1, \ldots, F_m$ from F (line 10) and examines whether each F_i is a party literal or a nested operator. In the second case, a fresh *virtual party* v_i is created and associated with F_i (the map V_{map} is used to keep track of this association). A virtual party is treated exactly as an actual party, except it is used only during this construction. The MSP for F is now created according to Lemma 4 and using both actual and virtual parties as the set $\mathcal{P}$. In the second part of the algorithm (lines 22–25) the MSPs for the nested operators (virtual parties v_i) are recursively created (line 23) and inserted in $\mathcal{M}$, according to Definition 7. The function ϕ related to the MSP $\mathcal{M}$, that maps a party to the rows they own, is used to get the row $\boldsymbol{r}_i$ of M that was labeled with v_i. Notice that in line 18, a fresh variable is created for each nested operator, so v_i owns a single row.

For the termination of the recursion, notice that, if F does not contain any nested threshold operators, V is the empty set when we reach line 22, and the algorithm returns. The next result therefore follows immediately from the definition of *insertion* and the fact that the algorithm starts with a 1×1 matrix.

Lemma 5. *Let F be an MBF that includes in total c operators in the form $\Theta_{d_i}^{m_i}$. The matrix M of the MSP constructed with Algorithm 2 has $m = \sum_1^c m_i - c + 1$ rows and $d = \sum_1^c d_i - c + 1$ columns.*

Lemma 5 implies that the resulting matrix M has size linear in the length of F. In the special case that each party appears only once in the access structure, M has n rows and at most n columns, where $n = |\mathcal{P}|$.

3.2.3 Checking for quorums

We now show how to determine whether a set constitutes a quorum using the MSP representation of the system and no other information

Algorithm 2 Construction of an MSP from a monotone Boolean formula F.

```
9: buildMSP(F)
10:     let Θ_d^m(F_1, ..., F_m) be the formula F
11:     R ← ∅
12:     V ← ∅
13:     V_map ← ∅
14:     for each F_i do
15:         if F_i is a literal p then
16:             R ← R ∪ {p}
17:         else
18:             declare v_i a new virtual party
19:             V ← V ∪ {v_i}
20:             V_map ← V_map ∪ {(v_i, F_i)}
21:     M ← Vandermonde-MSP(m, d, R ∪ V)
22:     for each v_i ∈ V do
23:         M_2 ← buildMSP(V_map(v_i))
24:         r_i ← φ(v_i)
25:         M ← M(r_i → M_2)
26:     return M
```

about the BQS (e.g., whether it is a threshold or a general BQS, or whether it was specified using threshold or other operators).

An MSP accepts a set A if and only if the rows of M_A span the vector $\boldsymbol{e}$, or, equivalently, the linear system $M_A^T\boldsymbol{x} = \boldsymbol{e}$ has solutions for $\boldsymbol{x}$. According to linear algebra, a necessary and sufficient condition for this is that the rank of M_A^T is equal to the rank of the augmented matrix $M_A^T|\boldsymbol{e}$. To check this condition, we perform Gaussian elimination on the augmented matrix $M_A^T|\boldsymbol{e}$ and bring it in row echelon form. If it contains a row with only zeros in the coefficient part but a nonzero value in corresponding constant part, then the rank of $M_A^T|\boldsymbol{e}$ is bigger that the rank of M_A^T, and A is not an authorized set. Otherwise, A is authorized (or superset of an authorized set).

Gaussian elimination has a cubic time complexity, so it is expensive to perform it every time we wish to check for a quorum. As an optimization we use the *PLU-decomposition* of matrix M^T, i.e., we calculate the $d \times d$ matrices P and L, and the $d \times m$ matrix U, such that $PM^T = LU$. Then, for any set A we get $PM_A^T = LU_A$, where P and L do not depend on A. In the initialization of the protocol we solve $L\boldsymbol{y} = P\boldsymbol{e}$ for $\boldsymbol{y}$,

where $\boldsymbol{y}$ is a d-vector. Then, instead of the equation $M_A^T \boldsymbol{x} = \boldsymbol{e}$ we can work with the equation $U_A \boldsymbol{x} = y$. In order to check whether a set A is authorized, we now have to bring $U_A|\boldsymbol{y}$ in row echelon form. Since U_A is an upper triangular matrix, some computational steps are avoided.

Notice here that it might be the case that A is a superset of an authorized group. These redundant parties can easily be identified from the echelon form, as they will correspond to the free variables of the system – variables whose corresponding column does not contain a pivot. Another situation worth to mention is that a party can own more than one rows of M. However, the algorithm described above also works in this case, since M_A will contain all rows owned by parties in A.

3.2.4 Concrete constructions of general Byzantine quorum systems

We now consider two specific families of general BQS that have been studied in the literature and show how they can be encoded as MSPs.

Attribute-based BQS. A BQS of this family is defined over a set of attributes, which are associated with the parties, and a quorum is described in terms of required attributes. Let $\mathcal{X} = \{\chi_1, \ldots, \chi_r\}$ denote the set of attributes and $\Psi \subseteq \mathcal{P} \times \mathcal{X}$ the relation between parties and attributes. We say that party p_j *holds* an attribute χ whenever $(p_j, \chi) \in \Psi$. An *attribute-based MBF* is a monotone Boolean formula $F(\chi_1, \ldots, \chi_r)$ over the attributes $\mathcal{X}$ and implements a BQS where a set $A \subseteq \mathcal{P}$ is a quorum whenever the attribute set $\{\chi \in \mathcal{X} \mid \exists p \in A : (p, \chi) \in \Psi\}$, collectively held by the parties in A, satisfies F. By adding one more syntactic rule, we can also specify the requirement that an attribute is held by at least a number of parties. Let each $\chi_i \in \mathcal{X}$ be related with L_i parties, i.e., $|\{p \in \mathcal{P} \mid (p, \chi_i) \in \Psi\}| = L_i$, and let $\ell_i \leq L_i$. Then, a formula $F(\chi_1^{(\ell_1)}, \ldots, \chi_r^{(\ell_r)})$ specifies that A is a quorum if, in addition to the aforementioned condition, each χ_i is held by at least l_i distinct parties, i.e., $|\{p \in A \mid (p, \chi_i) \in \Psi\}| \geq \ell_i$, for $1 \leq i \leq r$.

An MSP $\mathcal{M} = (M, \rho, \boldsymbol{e}_1, \mathcal{P})$ that implements $F(\chi_1^{(\ell_1)}, \ldots, \chi_r^{(\ell_r)})$ can be constructed as follows. First, an MSP $\mathcal{M}' = (M', \rho', \boldsymbol{e}'_1, \mathcal{F})$ is created for $F(\chi_1, \ldots, \chi_r)$, using the methods presented in the previous sections. Then an insertion $\mathcal{M}'(\chi_i \to \mathcal{M}_i)$ is performed for every χ_i, as described in Definition 7, where $\mathcal{M}_i$ is an MSP such that M_i is the $L_i \times \ell_i$ Vandermonde matrix and ρ_i is a function labelling the rows of M_i with the

parties related to χ_i. Notice that the resulting MSP $\mathcal{M}$ is defined on the set of parties $\mathcal{P}$ and not the set of attributes $\mathcal{X}$.

In Chapter 1 we mentioned an attribute-based BQS, where parties are organized in two dimensions, based on their location and operating system (OS). We now instantiate this BQS using this methodology. There are two families of attributes, location and OS. We use the attributes $\{\chi_{11}, \chi_{12}, \chi_{13}, \chi_{14}\}$ for the four different locations and the attributes $\{\chi_{21}, \chi_{22}, \chi_{23}, \chi_{24}\}$ for the four different OS. The 16 parties are arranged in a four by four grid, so that each party is related with exactly one attribute from each family. The system tolerates the simultaneous failure of all parties in one location and all parties with a specific OS. Thus, a set is a quorum if it contains at least three parties with different OS for at least three different locations. This BQS is implemented by the attribute-based MBF

$$\Theta_3^4\left(\chi_{11}^{(3)}, \chi_{12}^{(3)}, \chi_{13}^{(3)}, \chi_{14}^{(3)}\right) \wedge \Theta_3^4\left(\chi_{21}^{(3)}, \chi_{22}^{(3)}, \chi_{23}^{(3)}, \chi_{24}^{(3)}\right).$$

Following the method described above, a 4×3 Vandermonde matrix will be inserted in every $\chi_{ij}^{(3)}$ when creating the MSP, which, according to Lemma 5, will have dimensions 32×22.

The M-Grid BQS. Malkhi *et al.* [114] proposed the *M-Grid* system, a family of BQS where $n = k^2$ parties are arranged in a $k \times k$ grid and up to b parties are allowed to be Byzantine, with $b \leq (\sqrt{n+1})/2$. A quorum consists of any $\sqrt{b+1}$ rows and $\sqrt{b+1}$ columns. Actually, the M-Grid was proposed as a *Byzantine masking quorum system* [113], a category of BQS that requires a stronger intersection property than the Byzantine dissemination quorum systems, but one can adapt the construction accordingly.

For a dissemination BQS, the requirement for b is $b \leq k-1$ and a quorum consists of any t rows and t columns, where $t = \lceil \sqrt{b/2+1} \rceil$. To see this, notice that if two quorums Q_1 and Q_2 have a row or a column in common, then $|Q_1 \cap Q_2| \geq k \geq b+1$. Otherwise, the intersection of Q_1's columns with Q_2's rows is disjoint from the intersection of Q_2's columns with Q_1's rows, so $|Q_1 \cap Q_2| \geq 2\sqrt{b/2+1}\sqrt{b/2+1} > b+1$. In both cases, the *consistency* property of a BQS is satisfied.

To encode the M-Grid BQS we define the attribute set $\mathcal{X} = \{R_1, \ldots, R_k, C_1, \ldots, C_k\}$ and assign the party s_{ij} at row i and column j the attributes R_i and C_j. The attribute-based MBF related to this

BQS family is

$$\Theta_t^k\left(R_1^{(k)},\ldots,R_k^{(k)}\right)\wedge\Theta_t^k\left(C_1^{(k)},\ldots,C_k^{(k)}\right).$$

The formula has $3+2k$ threshold operators, considering the *and* operator as a 2-out-of-2 threshold and recalling that our method inserts a $k\times k$ MSP in the attributes $R_i^{(k)}$ and $C_j^{(k)}$. The resulting MSP that implements the M-Grid BQS has $2n$ rows and $2(n+t-k)<2n$ columns, by Lemma 5.

3.3 Consensus using general Byzantine quorums systems

HotStuff [163] is an efficient leader-driven Byzantine fault-tolerant state-machine replication (SMR) algorithm. The nodes that take part in the protocol are separated into *replicas*, which actually run the protocol, and *clients*, which submit requests to the replicas and receive totally-ordered responses. The trust assumptions are specified by the number of replicas n and the number of tolerated faults f. The replicas maintain a tree structure, whose *nodes* contain batches of clients' commands and get committed in a monotonically increasing way. Two nodes *conflict* if none of them extends from the other.

HotStuff is presented in three versions, the so-called *basic*, *chained*, and *implemented*. In the *basic* version, each view consists of four phases, called *prepare*, *pre-commit*, *commit*, and *decide*. In each phase, the leader waits for $n-f$ different vote messages from the replicas, constructs a *quorum certificate* (QC) upon receiving them, and starts the next phase by broadcasting this certificate to the replicas. The view changes in the end of the *decide* phase, or whenever the replicas time out waiting for a leader's message. Each view has a deterministically determined leader. The *chained* version pipelines the four phases into one *generic* phase. This serves as the prepare phase for the new node in the tree, as the pre-commit phase for the previous node and so on, so that the four phases map to four successively ordered requests. Finally, the implemented version presents further optimizations. The prototype implementation of threshold HotStuff, which we also use for generalized HotStuff, is based on the *implemented* version.

The generalized HotStuff protocol is *instantiated* with a Byzantine quorum system, which specifies its trust assumptions. The leader now

collects votes from a quorum of parties and constructs a QC by concatenating them. Upon receiving the QC, the replicas validate the signatures, as well as the fact that the voters indeed form a quorum. A quorum is also required to trigger a view change against a faulty leader.

The pseudocode of basic HotStuff with general BQS is presented in Algorithm 3 and 4. We give a brief description of the data structures used and refer to [163] for more details. A message consists of four fields, *type*, *viewNumber*, *node*, and *justify*. The *type* can be one of NEW-VIEW, PREPARE, PRE-COMMIT, COMMIT, DECIDE. The *viewNumber* is always populated with the current view number. The field *node* is used in the *prepare* phase by the leader to propose the new leaf node, as well as by replicas in vote messages. Finally, *justify* is always used by the leader to send a valid QC and by the replicas to send their *prepareQC* in a NEW-VIEW message. A vote message, sent by replicas, additionally contains a signature over the fields *type*, *viewNumber*, *node*. The QC data structure consists of four fields, *type*, *viewNumber*, *node*, and *sig*. The *type* can be one of PREPARE, PRE-COMMIT, COMMIT and is used to indicate the phase in which the votes used to construct the QC were cast. The fields *viewNumber* and *node* indicate the view in which the QC was created and the node it justifies, respectively. Finally, the field *sig* contains the signatures on the vote messages of the quorum that was used to construct the QC.

In the pseudocode we omit the details related to the signing and verification of the messages, the verification of a QC and the signing of the vote messages. We denote as p_ℓ the leader of a view. As in the original protocol, this could be any deterministic function from the view number to the replicas, as long as it eventually proposes a correct leader. If an interrupt happens when replicas are waiting for a message, line 69 is executed. The variables *new-views*, *prepare-votes*, *precommit-votes*, and *commit-votes*, used by the leader to store the votes until a quorum is received, are emptied in each view (not shown for brevity).

HotStuff works in the partial-synchrony model [66], where there is an unknown *Global Stabilization Time (GST)*, after which the communication between two correct replicas becomes synchronous. The safety of the HotStuff protocol as presented in [163] is based on the properties of threshold Byzantine quorum systems, namely the $n > 3f$ condition. In the generalized protocol the safety is reduced to the properties of the general BQS. The generalized version of HotStuff satisfies the same safety and liveness theorems as threshold HotStuff, which we now present and prove for the generalized case.

Algorithm 3 Basic HotStuff, code for party p_i, where p_ℓ is the epoch leader.

State:

27: $prepareQC \leftarrow \perp$; $lockedQC \leftarrow \perp$; $curView \leftarrow 1$

// *PREPARE* phase

28: **upon** receiving message [NEW-VIEW, $viewNumber, node, justify$] from p_j **such that** $viewNumber = curView - 1$ **do** // only p_ℓ

29: $new\text{-}views[j] \leftarrow justify$

30: **if exists** $\{p_k \in \mathcal{P} \mid new\text{-}views[k] \neq \perp\} \in \mathcal{Q}$ **then**

31: $V = \{new\text{-}views[k] \mid new\text{-}views[k] \neq \perp\}$

32: $highQC \leftarrow \text{argmax}_{v \in V}(v.viewNumber)$

33: $curProposal \leftarrow$ new node

34: $curProposal.parent \leftarrow highQC.node$

35: $curProposal.cmd \leftarrow$ client's command

36: send message [PREPARE, $curView, curProposal, highQC$] to all $p_j \in \mathcal{P}$

37: **upon** receiving message [PREPARE, $viewNumber, node, justify$] from p_ℓ **such that** $viewNumber = curView$ **do**

38: **if** $node$ extends from $justify.node$

39: **and** ($node$ extends from $lockedQC.node$

40: **or** $justify.viewNumber > lockedQC.viewNumber$) **then**

41: send vote message [PREPARE, $curView, node, \perp$] to p_ℓ

// *PRE-COMMIT* phase

42: **upon** receiving vote $v =$ [PREPARE, $viewNumber, node, justify$] from p_j **such that** $viewNumber = curView$ **do** // only p_ℓ

43: $prepare\text{-}votes[j] \leftarrow v$

44: **if exists** $\{p_k \in \mathcal{P} \mid prepare\text{-}votes[k] \neq \perp\} \in \mathcal{Q}$ **then**

45: $V = \{prepare\text{-}votes[k] \mid prepare\text{-}votes[k] \neq \perp\}$

46: $prepareQC \leftarrow QC(V)$

47: send message [PRE-COMMIT, $curView, \perp, prepareQC$] to all $p_j \in \mathcal{P}$

48: **upon** receiving message [PRE-COMMIT, $viewNumber, node, justify$] from p_ℓ **such that** $viewNumber = curView$ **and** $justify.type =$ PREPARE **do**

49: $prepareQC \leftarrow justify$

50: send vote message [PRE-COMMIT, $curView, justify.node, \perp$] to p_ℓ

Algorithm 4 Basic HotStuff, continued.

```
    // COMMIT phase
51: upon receiving vote v = [PRE-COMMIT, viewNumber, node, justify] from p_j
                    such that viewNumber = curView do          // only p_ℓ
52:     precommit-votes[j] ← v
53:     if exists {p_k ∈ P | precommit-votes[k] ≠⊥} ∈ Q then
54:         V = {precommit-votes[k] | precommit-votes[k] ≠⊥}
55:         precommitQC ← QC(V)
56:         send message [COMMIT, curView, ⊥, precommitQC] to all p_j ∈ P

57: upon receiving message [COMMIT, viewNumber, node, justify] from p_ℓ
                    such that viewNumber = curView
                    and justify.type = PRE-COMMIT do
58:     lockedQC ← justify
59:     send vote message [COMMIT, curView, justify.node, ⊥] to p_ℓ

    // DECIDE phase
60: upon receiving vote v = [COMMIT, viewNumber, node, justify] from p_j
                    such that viewNumber = curView do          // only p_ℓ
61:     commit-votes[j] ← v
62:     if exists {p_k ∈ P | commit-votes[k] ≠⊥} ∈ Q then
63:         V = {commit-votes[k] | commit-votes[k] ≠⊥}
64:         commitQC ← QC(V)
65:         send message [DECIDE, curView, ⊥, commitQC] to all p_j ∈ P

66: upon receiving message [DECIDE, viewNumber, node, justify] from p_ℓ
                    such that viewNumber = curView
67:                 and justify.type = COMMIT do
68:     output decide(justify.node)
69:     send message [NEW-VIEW, curView, ⊥, prepareQC] to p_{ℓ+1}
```

Theorem 6. *If w and b are conflicting nodes, they cannot be both decided, each by a correct replica.*

Proof. Let qc_1 and qc_2 be the valid certificates, with qc_1 created with the votes of a quorum Q_1 and qc_2 with the votes of a quorum Q_2, that convinced the two replicas to decide, that is $qc_1.type = \text{COMMIT}$, $qc_1.node = w$, $qc_2.type = \text{COMMIT}$, $qc_2.node = b$. Also, let $qc_1.viewNumber = v_1$ and $qc_2.viewNumber = v_2$. First note that v_1 and v_2 cannot be the same. That would mean that the votes in Q_1 and Q_2 were cast in the same view, which would require the replicas in $Q_1 \cap Q_2$ to vote twice in that view. But this is impossible, since Algorithms 3–4 allow replicas to vote only once in the *commit* phase $Q_1 \cap Q_2$ contains at least on correct replica. W.l.o.g. let $v_1 < v_2$ and let v_2 be the first view after v_1 for which a conflicting block is decided.

For qc_2 to be created there must first have been a valid *prepareQC* for node b. This could have been formed in view v_2 or in an earlier. Let v_s be the first view after v_1 in which a valid prepareQC qc_s was formed. So, $qc_s.type = \text{PREPARE}$, $qc_s.node = b$ and $qc_s.viewNumber = v_s$ and Q_s is a quorum of replicas, whose votes where used to create qc_s.

Consider now a replica r that voted for qc_1 and qc_s, i.e. $r \in Q_1 \cap Q_s$. During view v_1, r must had received a valid *precommitQC* and set it to its *lockedQC*, with $lockedQC.node = w$, before casting its vote for the *commitQC* qc_1. Let us examine now the *prepare* phase of view v_s, in which the leader proposed the new block b, and specifically the conditions in lines 39 and 40. By the minimality of v_s, r was still locked on *lockedQC* in that phase. By assumption b and w were conflicting nodes, so the condition in line 39 was FALSE. Moreover, *justify.viewNumber* was not larger than $lockedQC.viewNumber = u_1$, again by the minimality of v_s, because that would mean that a valid *prepareQC* was created in a view smaller than v_s. So the condition in line 40 was also FALSE. As a result, every replica in $r \in Q_1 \cap Q_s$ must be faulty. But this contradicts the quorum intersection property, thus such qc_1 and qc_2 cannot exist. □

Theorem 7. *After GST, there exists a bounded time period T_f such that if all correct replicas remain in view v during T_f and the leader for view v is correct, then a decision is reached.*

Proof. Assume a correct leader that collects NEW-VIEW messages from a quorums Q_1 of replicas. Let qc_l be the highest *lockedQC* among all replicas. There must be at least a quorum Q_2 of replicas that have

received (and voted for) a prepareQC qc_p that matches qc_l. By the quorum intersection property, $Q_1 \cap Q_2$ contains a non-empty set of non-faulty replicas, through which the leader will learn qc_p and use it as its *highQC* in the PREPARE message. Since all the correct replicas remain in view v, they will vote in all the phases and a decision will be reached. □

3.4 Implemented HotStuff

In Algorithms 5 and 6 we show the generalized *implemented* HotStuff, so as to document our changes with regard to [163].

The call *isQuorum*(*votes*[*b*]) checks whether the replicas in *votes*[*b*] constitute a quorum, using our algorithm described in 3.2.2. The function *getLeader* is not defined in HotStuff but is specified by the application. Procedure *onBeat* is also called by the leader in order to propose new clients' commands at points specified by the application.

3.5 Evaluation

We have implemented general BQS in HotStuff [163][1]. The new functionality has been added in the form of a C++ library into the existing code base. We use *nholmann-json* [108] to parse the user-defined quorum-specification file and Shoup's *NTL* [151] for linear algebra over $\mathbb{Z}_p$. As in the original version of HotStuff, our implementation uses secp256k1 for all signatures. The prototype code does not make use of threshold signatures, instead stores all the received votes for a block and verifies them independently. We keep the same logic for our generalized quorum votes.

Setup. In our evaluations, we report on benchmarks with four different versions of HotStuff that differ in the way how replicas and clients encode quorums. Their features are summarized in Table 3.1. In the original HotStuff algorithm (*Counting-All*), replicas and clients know the parameters n and f, the number of total replicas and failures, respectively, and determine whether they have received messages from a quorum by counting. In *MBF-All* the replicas and the clients are given the Byzantine quorum system, which can be a threshold or a general

[1] We used the prototype implementation available at https://github.com/hot-stuff/libhotstuff.

Algorithm 5 Implemented HotStuff, code for party p_i

```
    State
70:     prepareQC ←⊥; lockedQC ←⊥; curView ← 1

71: procedure createLeaf (parent, cmd, qc, height)
72:     b ← new node
73:     b.parent ← parent; b.cmd ← cmd
74:     b.justify ← qc; b.height ← height
75:     return b

76: procedure update (b*)
77:     b'' ← b*.justify.node
78:     b' ← b''.justify.node
79:     b ← b'.justify.node
80:     updateQCHigh(b*.justify)              //PRE-COMMIT phase on b''
81:     if b'.height > b_lock.height then     //COMMIT phase on b'
82:         b_lock ← b'
83:     if b''.parent = b' and b'.parent = b then    //DECIDE phase on b
84:         onCommit(b)
85:         b_exec ← b

86: procedure onCommit (b)
87:     if b_exec.height < b.height then
88:         onCommit(b.parent)
89:     execute(b.cmd)

90: procedure onReceiveProposal (m = [GENERIC, b_new, ⊥])
91:     if b_new.height > vheight and (b_new extends b_lock or
                b_new.justify.node.height > b_lock.height) then
92:         vheight ← b_new.height
93:         send message [GENERIC-VOTE, b_new, ⊥] to getLeader()
94:     update(b_new)

95: procedure onReceiveVote (m = [GENERIC-VOTE, b, ⊥]) from p_j
96:     votes[b] ← votes[b] ∪ {⟨j, m.sig⟩}
97:     if isQuorum(votes[b]) then
98:         qc ← QC({v_j}_{j=1}^k)
99:         updateQCHigh(qc)
```

Algorithm 6 Implemented HotStuff, continued.

100:**function** *onPropose* $(b_{\text{new}}, cmd, qc_{\text{high}})$
101: $b_{\text{new}} \leftarrow createLeaf(b_{\text{leaf}}, cmd, qc_{\text{high}}, b_{\text{leaf}}.height + 1)$
102: send message $[\text{GENERIC}, b_{\text{new}}, \perp]$ to all $p_j \in \mathcal{P}$
103: **return** b_{new}

104:**procedure** *updateQCHigh* (qc'_{high})
105: **if** $qc'_{\text{high}}.node.height > qc_{\text{high}}.node.height$ **then**
106: $qc_{\text{high}} \leftarrow qc'_{\text{high}}$
107: $b_{\text{leaf}} \leftarrow qc_{\text{high}}.node$

108:**procedure** *onBeat* (cmd)
109: **if** $i = getLeader()$ **then**
110: $b_{\text{leaf}} \leftarrow onPropose(b_{\text{leaf}}, cmd, qc_{\text{high}})$

111:**procedure** *onNextSyncView* (cmd)
112: send message $[\text{NEW-VIEW}, \perp, qc_{\text{high}}]$ to $getLeader()$

113:**procedure** *onReceiveNewView* $([\text{NEW-VIEW}, \perp, qc'_{\text{high}}])$
114: $updateQCHigh(qc'_{\text{high}})$

BQS, encoded as a monotone Boolean formula. Here we use Algorithm 1 to check for quorums. For *MSP-All*, replicas and clients are given an MSP-encoded BQS, again threshold or generalized, and use the algorithm of Section 3.2.2 to decide whether a set of parties is a quorum. According to the standard practice, replicas use batching to amortize various expensive operations (signatures and potentially Gaussian elimination) over multiple requests. However, the clients collect responses individually for every single request. This incurs a large cost that is not part of the replication protocol per se but is due to the way how clients produce requests and check for quorums. For this reason, we experiment also with a fourth protocol, called *MSP-Replicas*, where only the replicas use an MSP. In this setting, the clients are mapped to replicas. Since the replicas receive and verify batches of requests at once, there is no further need to perform the quorum check on individual requests.

The evaluation in the original HotStuff paper [163] uses a batch size of 400 because the latency of batching becomes higher than the cost of cryptographic operations with larger batches. Hence, we run all our experiments with batch size 400. Finally, we work only with the three-

Table 3.1. The evaluated protocols.

System	BQS implementation in replicas	BQS implementation in clients	Supported types of BQS
Counting-All	counting	counting	threshold
MBF-All	MBF	MBF	threshold & generalized
MSP-All	MSP	MSP	threshold & generalized
MSP-Replicas	MSP	counting	threshold & generalized

phase HotStuff.

We use VMs on a leading cloud provider, with each replica or client running on a single VM with 16 vCPUs (Intel Xeon Broadwell, 2.6 GHz, or Intel Xeon Skylake, 2.7 GHz), 32 GB RAM, and SSD local storage. We use a varying number of VMs – up to 40 replicas and 32 clients. All experiments are done over the LAN inside one data center, with a RTT of less than 1 ms. As this setup eliminates most network delays, it exposes the overhead added by the general BQS code. For the same reason, we use only zero-sized request and response payloads. In realistic deployments (on a wide-area network and with significant payload data), the extra cost of general quorums would be less visible. All measurements are made on the client. Finally, the maximum available bandwidth among the VMs was measured by *iperf* as 1–2 Gbits per second.

Throughput vs. latency. We first measure throughput and latency in a small system with four replicas, with the goal of comparing the behavior of the four different quorum-system implementations. We use a threshold BQS because all four protocols can be instantiated with it, that is, in Counting-All, this is specified by two numbers, $n = 4$ and $f = 1$, in MBF-All by the $\Theta_3^4(\mathcal{P})$ MBF, and in the last two protocols by an MSP implementing the $\Theta_3^4(\mathcal{P})$ access structure. The reported values were produced by first fixing the request rate per client and increasing the number of clients from one to eight and then, with the number of clients fixed at eight, increasing the request rate even further for each of them, until the system saturates. The result is depicted in Figure 3.3.

All four protocols exhibit similar behavior. Counting-All saturates at 188.4K tx/sec, followed by MBF-All at 179.3K tx/sec, which is less than 5% lower. The peak throughput of MSP-based protocols are slightly lower. Specifically, MSP-Replicas delivers 175.5K tx/sec before sat-

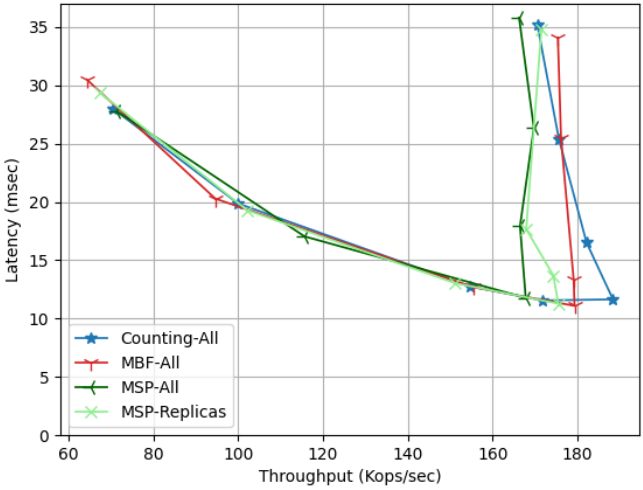
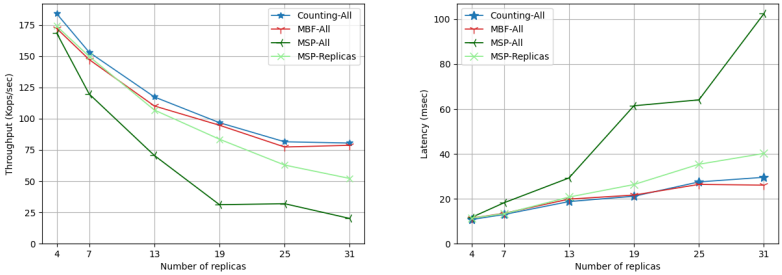


Figure 3.3. Throughput vs. latency for 1–8 clients and different implementations of the threshold BQS with four replicas.

uration, which translates to an overhead of almost 7% compared to Counting-All, while MSP-All reaches roughly 167.8K tx/sec, for an overhead of 11%. The latency at the saturation point is about 11.5ms for all protocols. We conclude that in a small system like this, with four parties, generalizing a protocol does not significantly impact its efficiency.

Scalability. In this evaluation we measure the throughput and latency in a system with a varying number of replicas. We use $n = 3f + 1$ replicas, for $f \in \{1, \ldots, 10\}$, and a varying number of clients. The trust assumption is again a threshold quorum system with n replicas, of which up to f may fail, specified in the appropriate way for each system. For each n we increase the request rate per client and report the throughput and latency just before saturation. The question we want to answer is how the generalized protocols (MBF-All, MSP-All, MSP-Replicas) scale in comparison to Counting-All. The results are shown in Figure 3.4a (throughput) and Figure 3.4b (latency).

We notice that Counting-All and MBF-All scale up almost identi-



(a) Throughput **(b)** Latency

Figure 3.4. Scalability of the four protocols when instantiated with a threshold BQS.

cally. In a system with 31 replicas they achieve a throughput of 80.5K and 78.7K tx/sec, respectively, with latencies of 29.6ms and 26ms. MSP-Replicas achieves throughput and latency very similar to Counting-All for low values of n and comparable to Counting-All for higher n. At $n = 13$ the throughput of MSP-Replicas is 9% lower than that of Counting-All, while the latency is only 4% higher. With $n = 31$, throughput and latency of MSP-Replicas lie both approximately 35% behind the numbers for Counting-All. We conclude that the overhead added by the MSP-based quorum-checking code is relatively small for the replicas, considering all the other tasks they have to carry out, such as signature evaluation and message processing, especially when batching is used. However, the protocol where both the replicas and the clients use MSPs does not scale so well. This is because clients do not use batching but operate on the MSP matrix for every received response. Moreover, in the original HotStuff prototype implementation, the clients do not verify the signatures on the response messages at all (!) and therefore, this operation is very fast and lets the overhead of the MSP appear large. With signature verification enabled, as in a production system, additional cost incurred by the MSP representation would be much less visible.

Scalability with general Byzantine quorum systems. We now evaluate the protocols beyond threshold BQS. The question we want to answer with this benchmark is how they scale when instantiated with

a general BQS, in comparison to when instantiated with a threshold BQS. We focus on MBF-All and MSP-Replicas, which perform best in the previous experiments, and run them on two different families of BQS. The first is the 2-layered-1-common general BQS presented in Section 3.2, and the second is a threshold BQS. For 2L1C we vary the parameter k from 4 to 10, resulting in a system with $4k$ parties, while the threshold BQS is specified by the MBF $\Theta^n_{\lceil \frac{2n+1}{3} \rceil}(p_1, \ldots, p_n)$, for $n = 4k$. We do not consider Counting-All in this benchmark because it cannot be instantiated with the general BQS.

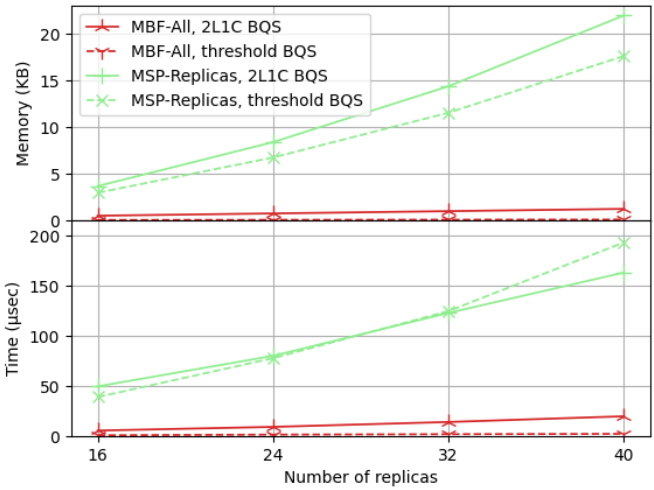
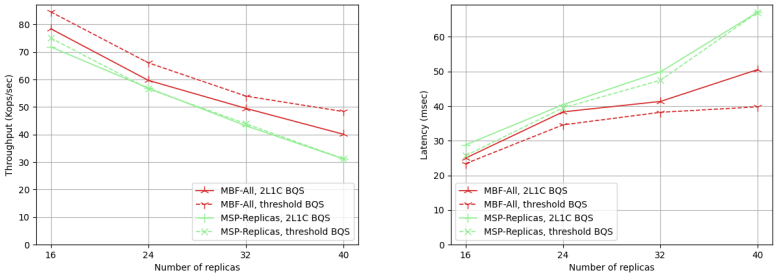


Figure 3.5. Memory and time required to store the BQS and check for quorums for the MSP-based and MBF-based implementations, when instantiated with two different trust assumptions, the generalized 2L1C for $k = 4, \ldots, 10$, resulting in $4k$ parties, and the 2/3 Byzantine threshold on a set of $4k$ parties.

We first report a direct comparison between the MBF method and the MSP method for encoding a general BQS. In Figure 3.5 we show the memory required by each replica to store the BQS specification and the average time needed to check whether a set (chosen uniformly at random and repeated 10000 times) is a quorum, based on our implementation.

Both MBF-All and MSP-Replicas are considered, instantiated with both the 2L1C and the threshold BQS. The MBF-based encoding is far more efficient than the MSP implementation, both in terms of memory consumption and evaluation time.

In Figures 3.6a and 3.6b we report the throughput and latency, respectively. In this experiment we run two replica instances in every VM, so the values reported here are overall lower than in the previous benchmarks. The performance of MSP-Replicas when running with the generalized and the threshold quorum specifications is similar. This is because in both cases the replicas have to perform Gaussian elimination on matrices of comparable dimensions. MBF-All also scales in a similar way for both families of trust assumptions, but this benchmark shows that its efficiency is slightly affected by the specified BQS. This is, first, because general BQS are implemented by longer monotone Boolean formulas, but also because general BQS have a (sometimes much) smaller number of quorums than threshold BQS, which might affect the leader when waiting for a quorum of votes. It is worth to mention that the MBF-based protocols perform better than the MSP-based ones also in this benchmark.



(a) Throughput **(b)** Latency

Figure 3.6. Scalability of MBF-All and MSP-Replicas when running under two different trust assumptions, the generalized 2L1C for $k = 4, \ldots, 10$, resulting in $4k$ parties, and the 2/3 Byzantine threshold on a set of $4k$ parties.

3.6 Discussion

In this chapter we have seen how general Byzantine quorum systems (BQS) can be efficiently encoded as a monotone Boolean formula (MBF) and as a monotone span program (MSP), and how they can be used in a consensus algorithm.

Our benchmarks illustrate the added value of general BQS and demonstrate that they have small overhead. One can therefore specify complex, non-threshold trust assumptions in consensus protocols without significantly sacrificing efficiency. The MBF-based protocol performs consistently better than the MSP-based, which can be expected due to the higher implementation complexity. The performance of the MBF-based protocol was identical or comparable to the original threshold HotStuff, although it can be slightly affected by the complexity of the BQS, since more complex trust assumptions result in longer formulas. The protocol where both the replicas and the clients use the MSP does not scale well and can only be used in small systems. Nonetheless, in applications where all the nodes participate in the protocol, i.e., clients are not disjoint from servers, encoding the BQS as an MSP also results in high efficiency, as was shown by MSP-Replicas in the benchmarks.

We anticipate that our work will pave the way for more protocols generalizing threshold trust assumptions. This can be combined with the novel ideas presented in the BFT literature, e.g., the combination of crash and Byzantine faults [36] or with peer-to-peer gossip [30].

Chapter 4

Composition of general Byzantine quorum systems

This chapter defines the notion of *quorum composition* and presents composition rules for general Byzantine quorums in what is known as the *symmetric trust* setting, where one global fail-prone system (FPS) and one Byzantine quorum system (BQS) exist and are accepted by all parties. The corresponding full paper [9] extends these rules to the *asymmetric trust* setting, where each party specifies its own FPS and BQS. For that setting, the authors first define the notion of a *tolerated system*, which abstracts the failures that are tolerated in a system with asymmetric trust, and then apply the symmetric rules, as presented in this chapter, on tolerated systems.

4.1 Introduction

In this chapter we study the problem of composing trust assumptions, expressed in terms of Byzantine quorum systems. Starting from two or more running distributed systems, each one with its own assumption, how can they be combined, so that their participant groups are joined and operate together? A simple, but not so intriguing solution could be to stop all running protocols and to redefine the trust structure from

scratch. One drawback is that the composite system would have to be restarted. Moreover, one needs to ensure that the combined system satisfies the liveness and safety conditions, as expressed by the Q^3-condition. This work presents methods for assembling trust assumptions from different, intersecting or disjoint, sets of processes. Our methods describe the resulting fail-prone system and the corresponding quorum system.

Contributions. Specifically, the contributions of this work are as follows:

1. We formulate the task of composing Byzantine quorum systems and propose properties that any such a composition rule should satisfy.
2. We show how to compose two or more systems, in a way that administrators or processes in one system do not need to make new assumptions about those in the other.
3. Our composition rules guarantee that *consistency* and *availability* will be satisfied in the composite quorum system.

Organization. This chapter is structured as follows. In Section 4.2 we review related work. In Section 4.3 define quorum composition and its desired properties, and in Section 4.4 we show different composition rules for both fail-prone systems and quorum systems. These rules achieve different properties, which we explore formally. In Section 4.5 we discuss and conclude this chapter.

System model. We consider a system $\mathcal{P}$ with an arbitrary number of *processes* p_i that communicate with each other. A protocol for $\mathcal{P}$ consists of a collection of programs with instructions for all processes.

An *execution* starts with all processes in a special initial state; subsequently the processes repeatedly change their state through computation steps. Every execution is fair in the sense that, informally, processes do not halt prematurely when there are still steps to be taken.

An *honest* process follows its protocol during an execution, and a *corrupted* process may crash or deviate arbitrarily from its specification, e.g., when corrupted by the adversary. We assume for simplicity that the corrupted processes fail right at the start of an execution.

4.2 Related work

Byzantine quorum systems (BQS) have originally been formalized by Malkhi and Reiter [113] to generalize classical quorum systems toward processes prone to Byzantine failures. They model symmetric trust, where every process in the system adheres to a global, common assumption. Many distributed protocols employ BQS as their foundation; in the area of state-machine replication, for example, they range from PBFT [43] to Tendermint [30], HotStuff [163], and other blockchain-specific protocols. Recently, also general BQS have been demonstrated for implementing consensus [5].

Measures of quality for classical (non-Byzantine) quorum system have been studied by Naor and Wool [126] in terms of the load, capacity, and availability properties. The load (the probability of access of the busiest process) and availability (probability of some quorum surviving failures) properties have then been considered by Malkhi *et al.* [114] in the context of the Byzantine quorum systems. They construct different types of Byzantine quorum systems with optimal load or availability.

Subsequent literature extends the BQS model, seeking to overcome some limitations and to take them into practice. To this end, *probabilistic* quorum systems have been introduced by Malkhi *et al.* [115] as a tool for ensuring consistency of replicated data with high probability despite both benign and Byzantine failure of processes. They define the *ϵ-intersecting quorum systems* by relaxing the intersection property of a quorum system in a way that every two quorums fail to intersect with some small probability ϵ. By the quality measures, these new quorums show an improvement over the classic and Byzantine ones.

Alvisi *et al.* [11] introduce *dynamic* Byzantine quorum systems in the context of quorum-based Byzantine fault-tolerant data services. They present protocols for dynamically changing the threshold of the system. In this this way, they solve an intrinsic limitation of standard Byzantine quorums, which is their dependence on *a-priori* defined resilience thresholds.

Malkhi *et al.* [112] define *flexible Byzantine quorums* that allow processes in the system to have different faults models. Their work presents a new approach for designing Byzantine fault-tolerant consensus protocols which guarantees higher resilience by introducing a new *alive-but-corrupt* fault type, which denotes processes that attack safety but not liveness.

With the rise of blockchains, protocols using flexible trust structures have been deployed in practice as well. Ripple (www.ripple.com) and Stellar (www.stellar.org) do not base their resilience guarantees on a global threshold, but allow processes to express their own beliefs.

A related form of recursive composition of Byzantine quorum systems has been explored and utilized in the literature. The idea is that, given two systems, each occurrence of a process in the first is *replaced* by a copy of the second system. Malkhi *et al.* [114] construct and study composite BQS, such as *recursive threshold* BQS, using this idea. Hirt and Maurer [91] use this technique to reason about multiparty computation over access structures. Our approach is orthogonal to these works, in the sense that it places the two original systems on the same level. In other words, we explore the failures that two systems can tolerate when they are joined together, as opposed when one is inserted into the other.

4.3 Defining composition for general Byzantine quorum systems

Given two Byzantine quorum systems $\mathcal{Q}_1$ defined on processes $\mathcal{P}_1$ with fail-prone system $\mathcal{F}_1$, and $\mathcal{Q}_2$ defined on processes $\mathcal{P}_2$ with fail-prone system $\mathcal{F}_2$, we want to provide a *composition* rule between the two that allows the resulting BQS $\mathcal{Q}_3$ defined on processes $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$ with fail-prone system $\mathcal{F}_3$ to run a distributed protocol together. The resulting system should satisfy the consistency and availability properties of a BQS, that is, it should remain consistent and live in any execution where a fail-prone set in $\mathcal{F}_3$ fails.

In this work we explore the composition of two BQS as a means to allow them jointly run a protocol, *without* requiring the processes in one BQS to make new trust assumptions about the processes in the other. This is useful in practice because remodeling trust from scratch would be a manual and uncertain process. We do not consider the composition as a way to increase their resilience. For example, joining four singleton BQS will result in a system with four processes, none of which is expected to fail. This makes sense if one starts from the trust assumptions of singleton BQS; by definition, the single process it contains never fails. There could be other ways to compose the BQS, but they would require changing the assumptions of each individual BQS and it is subject of future work.

According to the previous discussion, we now state properties that we expect any form of composition for BQS should satisfy. Characterizing the failures the composite BQS can tolerate now becomes the challenge because multiple definitions of $\mathcal{F}_3$ are plausible. We want to ensure the following *properties*:

1. Any $B \in \mathcal{F}_3$ satisfies $B|_{\mathcal{P}_1} \in \mathcal{F}_1^*$, i.e., the failure of B is tolerated in the first system.
2. Any $B \in \mathcal{F}_3$ satisfies $B|_{\mathcal{P}_2} \in \mathcal{F}_2^*$, i.e., the failure of B is tolerated in the second system.
3. $\mathcal{F}_3$ satisfies the Q^3-condition.
4. For any $B \in \mathcal{F}_3$, there exists a $Q \in \mathcal{Q}_3$, a quorum in the composite system, such that $B \cap Q = \emptyset$, i.e., there is always a quorum consisting only of honest processes.

In the text above, the notation $\mathcal{X}|_{\mathcal{P}}$ denotes the restriction of a set $\mathcal{X}$ to $\mathcal{P}$.

We need properties 1 and 2 because, as we shall see next, they imply Property 3, and, hence, ensure consistency for the composite BQS against any fail-prone set in $\mathcal{F}_3$. Moreover, they enable a composition by using the existing assumptions, without requiring a redesign of the two systems. One might also desire that the inverse of properties 1 and 2 be satisfied, i.e., that any fail-prone set in $\mathcal{F}_1$ and $\mathcal{F}_2$ be tolerated in $\mathcal{F}_3$. However, we will later see that this does not always result in a BQS (i.e., in a fail-prone system that satisfies the Q^3-condition). Thus, the objective of a composition rule is to satisfy these properties, thus ensuring safety, while producing a maximal fail-prone system $\mathcal{F}_3$ (in the sense that it contains the largest fail-prone sets that could be created without having to redefine the trust assumptions within the original systems). Finally, the composition rule should also satisfy Property 4, which ensures liveness in the composite system.

Lemma 8. *Properties 1 and 2 above imply Property 3.*

Proof. Let us assume that $Q^3(\mathcal{F}_1)$ and $Q^3(\mathcal{F}_2)$. Towards a contradiction, let $F_A, F_B, F_C \in \mathcal{F}_3$ such that $F_A \cup F_B \cup F_C = \mathcal{P}_3$. Now consider the restriction of F_A, F_B and F_C to $\mathcal{P}_1$ (and similarly to $\mathcal{P}_2$). We have that $F_A|_{\mathcal{P}_1} \cup F_B|_{\mathcal{P}_1} \cup F_C|_{\mathcal{P}_1} = \mathcal{P}_1$. However, from Property 1, the sets $F_A|_{\mathcal{P}_1}$, $F_B|_{\mathcal{P}_1}$, and $F_C|_{\mathcal{P}_1}$ are each (subsets of) fail-prone sets in $\mathcal{F}_1$. We thus have found three fail-prone sets that cover $\mathcal{P}_1$, a contradiction to $\mathcal{F}_1$ satisfying the Q^3-condition. □

4.4 Composition rules for general Byzantine quorum systems

We now proceed to specific constructions. In the following, we present three composition methods of increasing suitability and give examples to show their weaknesses and strengths.

Construction 9 (Union composition). Let $\mathcal{Q}_1$ be a BQS defined on processes $\mathcal{P}_1$ with fail-prone system $\mathcal{F}_1$, and $\mathcal{Q}_2$ a BQS defined on processes $\mathcal{P}_2$ with fail-prone system $\mathcal{F}_2$, where $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$. The *union composition* of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ is a system defined on processes $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$ with fail-prone system

$$\mathcal{F}_3 = \mathcal{F}_1 \cup \mathcal{F}_2.$$

We can easily verify that the previous definition, given that $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, fulfills Properties 1 and 2. Thus, $\mathcal{F}_3$ satisfies the Q^3-condition and there exists a BQS $\mathcal{Q}_3$ with fail-prone system $\mathcal{F}_3$.

Lemma 10. *Given $\mathcal{F}_3$ as in Construction 9, a BQS $\mathcal{Q}_3$ is*

$$\mathcal{Q}_3 = \{Q_i \cup Q_j \mid Q_i \in \mathcal{Q}_1, Q_j \in \mathcal{Q}_2\},$$

with $\mathcal{Q}_1$ and $\mathcal{Q}_2$ BQS.

Proof. We first show that consistency property holds. So, for every $Q_1, Q_2 \in \mathcal{Q}_3$ such that $Q_1 = Q_i \cup Q_j$ and $Q_2 = Q_i^{'} \cup Q_j^{'}$, with $Q_i, Q_i^{'} \in \mathcal{Q}_1$ and $Q_j, Q_j^{'} \in \mathcal{Q}_2$, and for every $F \in \mathcal{F}_3$, with $F \in \mathcal{F}_1$ or $F \in \mathcal{F}_2$, we have $Q_1 \cap Q_2 = (Q_i \cup Q_j) \cap (Q_i^{'} \cup Q_j^{'})$, which equals $(Q_i \cap Q_i^{'}) \cup (Q_j \cap Q_j^{'})$, because $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$. By assumption, both $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are BQS. This means that, if $F \in \mathcal{F}_1$, then $\mathcal{Q}_i \cap Q_i^{'} \not\subseteq F$, and if $F \in \mathcal{F}_2$, then $\mathcal{Q}_j \cap Q_j^{'} \not\subseteq F$. The property then follows. Finally, the availability property follows from the fact that $\mathcal{P}_1$ and $\mathcal{P}_2$ are disjoint and $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are BQS. □

However, the fail-prone system obtained by Construction 9 results in a fail-prone system that tolerates only a few failures, namely those tolerated in each of the two original systems, and not any combination of them. Moreover, it would not work if $\mathcal{P}_1$ and $\mathcal{P}_2$ had any processes in common. The next notion moves towards a composition that tolerates any combination of failures that would be tolerated in the original systems.

Construction 11 (Cartesian composition on disjoint sets). Let $\mathcal{Q}_1$ be a BQS defined on processes $\mathcal{P}_1$ with fail-prone system $\mathcal{F}_1$, and $\mathcal{Q}_2$ a BQS defined on processes $\mathcal{P}_2$ with fail-prone system $\mathcal{F}_2$, where $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$. Then the *Cartesian composition* of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ is defined on processes $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$ and tolerates the failure of any combination of fail-prone sets of the original BQS. Formally,

$$\mathcal{F}_3 = \{F_i \cup F_j \mid F_i \in \mathcal{F}_1, F_j \in \mathcal{F}_2\}.$$

Lemma 12. *If $Q^3(\mathcal{F}_1)$ and $Q^3(\mathcal{F}_2)$, then for the fail-prone system $\mathcal{F}_3$ according to Construction 11, $Q^3(\mathcal{F}_3)$.*

Proof. Any $B \in \mathcal{F}_3$ satisfies $B|_{\mathcal{P}_1} \in \mathcal{F}_1$ and $B|_{\mathcal{P}_2} \in \mathcal{F}_2$, since $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$. Hence, the composition in Definition 11 satisfies the Properties 1 and 2, and, by Lemma 8, also $Q^3(\mathcal{F}_3)$. □

The previous lemma implies the existence of a BQS $\mathcal{Q}_3$ with fail-prone system $\mathcal{F}_3$. Such a $\mathcal{Q}_3$ can be obtained, as earlier, by

$$\mathcal{Q}_3 = \{Q_i \cup Q_j \mid Q_i \in \mathcal{Q}_1, Q_j \in \mathcal{Q}_2\}.$$

It is easy to show, in a similar way as in Lemma 10, that this $\mathcal{Q}_3$ satisfies consistency and availability properties. Moreover, if $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are canonical, $\mathcal{Q}_3$ will be the canonical BQS for $\mathcal{F}_3$.

Example 1. Let us consider the threshold case. Suppose $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be two BQS, defined on $\mathcal{P}_1$ and $\mathcal{P}_2$, where $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, containing 7 and 10 processes, and tolerating the failure of any 2 and 3 processes, respectively. This means that the first fail-prone system contains $\binom{7}{2} = 21$ sets of processes and the second fail-prone system contains $\binom{10}{3} = 120$ sets. Because in this work we join systems with already existing failure assumptions, we refrain from changing these assumptions for the composite system. Nevertheless, according to Lemma 12, the Cartesian product of the fail-prone systems leads to a fail-prone system where the Q^3-condition holds, assuming that the starting systems both satisfy the Q^3-condition and are disjoint.

We apply Construction 11 here, observing that the Q^3-condition is the generalization of the condition $n > 3f$ for the threshold case. As a result we obtain an assumption on 17 processes, which tolerates the failure of 5 processes, where 2 processes are from $\mathcal{P}_1$ and 3 from $\mathcal{P}_2$. More formally, the failure of a set F is tolerated in the composite system if and only if $|F \cap P_1| \leq 2 \wedge |F \cap P_2| \leq 3$.

This gives a total of 2520 possible tolerated subsets. Observe that $\mathcal{Q}_3$ is not a threshold BQS any more, and this was intended. A threshold BQS made of 17 processes would tolerate the failure of any 5 processes, which would lead to a total of $\binom{17}{5} = 6188$ fail-prone sets.

Example 2. We now show how Construction 11 fails to create a BQS $\mathcal{Q}_3$ if $\mathcal{P}_1$ and $\mathcal{P}_2$ intersect, because the Q^3-condition may not hold in the composite system. Let $\mathcal{Q}_1$ defined on $\mathcal{P}_1 = \{a, b, c, d, e\}$ with fail-prone system $\mathcal{F}_1 = \{\{a\}, \{b, c\}, \{d\}, \{c, e\}\}$ and $\mathcal{Q}_2$ defined on $\mathcal{P}_2 = \{d, e, f, g, h\}$ with fail-prone system $\mathcal{F}_2 = \{\{d\}, \{e\}, \{f, g\}, \{h\}\}$.

It is easy to verify that the Q^3-condition is satisfied in $\mathcal{Q}_1$ and $\mathcal{Q}_2$. We also see that, according to Construction 11, $\mathcal{Q}_3$ with processes $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$ contains, among others, the fail-prone sets $\{a, f, g\}, \{b, c, h\}, \{c, e, d\}$, which cover $\mathcal{P}_3$. Consequently, $\mathcal{Q}_3$ is not a BQS.

Example 2 shows that the Cartesian composition among fail-prone systems does not lead to a fail-prone system where the Q^3-condition holds, if the two systems have common processes. To overcome this issue, we introduce a third construction.

Definition 8. Let $\mathcal{A} = \{A_1, \ldots, A_m\}$ and $\mathcal{B} = \{B_1, \ldots, B_n\}$ be two sets of subsets of $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. We define $\mathcal{A} \otimes \mathcal{B}$ as the set that contains the union of all sets $A_i \in \mathcal{A}^*$ and $B_j \in \mathcal{B}^*$, under the restriction that either both A_i and B_j contain exactly the same subset of the processes common to $\mathcal{P}_1$ and $\mathcal{P}_2$ or they do not have anything in common. Formally,

$$\mathcal{A} \otimes \mathcal{B} = \big\{ A_i \cup B_j \mid A_i \in \mathcal{A}^* \wedge B_j \in \mathcal{B}^* \wedge (\forall C \subseteq \mathcal{P}_1 \cap \mathcal{P}_2 : C \subseteq A_i \Leftrightarrow C \subseteq B_j) \big\}.$$

Construction 13 (Cartesian composition). Let $\mathcal{Q}_1$ be a BQS defined on processes $\mathcal{P}_1$ with fail-prone system $\mathcal{F}_1$ and $\mathcal{Q}_2$ a BQS defined on processes $\mathcal{P}_2$ with fail-prone system $\mathcal{F}_2$, where $\mathcal{P}_1$ and $\mathcal{P}_2$ might contain common processes. Then the *composition* of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ is defined on $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$ and tolerates the failure of any combination of any fail-prone set (or subset of it) of the first system and any fail-prone set (or subset) of the second system, such that both contain exactly the same subset of the common processes. Formally,

$$\begin{aligned} \mathcal{F}_3 &= \mathcal{F}_1 \otimes \mathcal{F}_2 \\ &= \big\{ F_i \cup F_j \mid F_i \in \mathcal{F}_1^* \wedge F_j \in \mathcal{F}_2^* \wedge (\forall C \subseteq \mathcal{P}_1 \cap \mathcal{P}_2 : C \subseteq F_i \Leftrightarrow C \subseteq F_j) \big\}. \end{aligned}$$

The rule of Construction 13 states that any fail-prone set in $\mathcal{F}_3$ is of the form $F_i \cup F_j$, where F_i and F_j are fail-prone sets (or subsets of fail-prone sets) that either do not have any processes in common or, if they do, both contain exactly the same subset of $\mathcal{P}_1 \cup \mathcal{P}_2$. We demand $F_i \in \mathcal{F}_1^*$ and $F_j \in \mathcal{F}_2^*$, instead of $F_i \in \mathcal{F}_1$ and $F_j \in \mathcal{F}_2$, in order to construct a maximal $\mathcal{F}_3$, in the sense that it contains the maximal fail-prone sets that satisfy Properties 1 and 2.

Lemma 14. *If $Q^3(\mathcal{F}_1)$ and $Q^3(\mathcal{F}_2)$, then $Q^3(\mathcal{F}_3)$, with $\mathcal{F}_3$ as in Construction 13.*

Proof. Any $B \in \mathcal{F}_3$ either does not contain a set of common processes C among $\mathcal{P}_1$ and $\mathcal{P}_2$ or it does. In the former case, it is immediate to see that $B|_{\mathcal{P}_1} \in \mathcal{F}_1^*$ and $B|_{\mathcal{P}_2} \in \mathcal{F}_2^*$. In the latter case, B has been created as the union between $F_i \in \mathcal{F}_1^*$ and $F_j \in \mathcal{F}_2^*$, both containing the same subset of $P_1 \cap P_2$, according to Construction 13. It is thus not possible that a new element of $\mathcal{P}_1$ appears in $B|_{\mathcal{P}_1}$ that was not already in F_i, and similarly that a new element of $\mathcal{P}_2$ appears in $B|_{\mathcal{P}_2}$ that was not already in F_j. This implies that $B|_{\mathcal{P}_1} \in \mathcal{F}_1^*$ and $B|_{\mathcal{P}_2} \in \mathcal{F}_2^*$, and from Lemma 8 we get $Q^3(\mathcal{F}_3)$. □

Lemma 15. *Given $\mathcal{F}_3$ as in Construction 13, a BQS $\mathcal{Q}_3$ is*

$$\mathcal{Q}_3 = \{Q_i \cup Q_j \mid Q_i \in \mathcal{Q}_1, Q_j \in \mathcal{Q}_2\},$$

with $\mathcal{Q}_1$ and $\mathcal{Q}_2$ BQS.

Proof. Consistency and availability properties of $\mathcal{Q}_3$ can be proved in a similar way as Lemma 10, assuming $\mathcal{Q}_1$ and $\mathcal{Q}_2$ to be BQS. In fact, as in Lemma 10, we have that for every $Q_1, Q_2 \in \mathcal{Q}_3$, such that $Q_1 = Q_i \cup Q_j$ and $Q_2 = Q_i^{'} \cup Q_j^{'}$, with $Q_i, Q_i^{'} \in \mathcal{Q}_1$ and $Q_j, Q_j^{'} \in \mathcal{Q}_2$, we have $Q_1 \cap Q_2 = (Q_i \cup Q_j) \cap (Q_i^{'} \cup Q_j^{'})$, which results in $(Q_i \cap Q_i^{'}) \cup (Q_i \cap Q_j^{'}) \cup (Q_j \cap Q_i^{'}) \cup (Q_j \cap Q_j^{'})$. If $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, it is trivial to prove the result. Otherwise, given $F \in \mathcal{F}_3$ with $F = F_i \cup F_j$, where $F_i \in \mathcal{F}_1^* \wedge F_j \in \mathcal{F}_2^* \wedge \forall C \subseteq \mathcal{P}_1 \cap \mathcal{P}_2 : C \subseteq F_i \Leftrightarrow C \subseteq F_j$, we have two cases. If there are no common processes between F_i and F_j, then observe that F_i is contained in $\mathcal{F}_i^*$ and it is then a subset of a fail-prone set $\overline{F}_i$ in $\mathcal{F}_i$. The same happens for F_j. By assumptions, $\mathcal{Q}_1$ (respectively, $\mathcal{Q}_2$) are BQS. It follow that, $(Q_i \cap Q_i^{'})$ (respectively, $(Q_j \cap Q_j^{'})$) is not a proper subset of $\overline{F}_i$ and consequently of F_i (respectively of F_j). The result follows. The same reasoning can be applied if F_i and F_j contain a common subset $C \subseteq \mathcal{P}_1 \cap \mathcal{P}_2$. □

Example 3. Let us consider again the threshold case, where $\mathcal{Q}_1$ is defined on processes $\mathcal{P}_1 = \{a, b, c, d, e, f, g\}$ and $\mathcal{Q}_2$ on $\mathcal{P}_2 = \{g, h, i, j, k, l, m, n, o, p\}$. According to Construction 13, any two processes in $\mathcal{P}_1$ together with any three processes in $\mathcal{P}_2$ are tolerated to fail, because these failures would be tolerated in the original systems. However, if g together with any other process in $\mathcal{P}_1$ fails, then only two more failures in $\mathcal{P}_2$ are tolerated, because $g \in \mathcal{P}_2$ has already failed in the first system.

Example 4. Let $\mathcal{Q}_1$ be defined on processes $\mathcal{P}_1 = \{a, b, c, d, e\}$ and with fail-prone system $\mathcal{F}_1 = \{\{a\}, \{b, c\}, \{d\}, \{c, e\}\}$ and $\mathcal{Q}_2$ be defined on processes $\mathcal{P}_2 = \{d, e, f, g, h\}$ with fail-prone system $\mathcal{F}_2 = \{\{d\}, \{e\}, \{f, g\}, \{h\}\}$. Then, according to Construction 13 processes in $\mathcal{P}_3 = \{a, b, c, d, e, f, g, h\}$ have fail-prone system

$$\mathcal{F}_3 = \{\{a, f, g\}, \{a, h\}, \{b, c, f, g\}, \{b, c, h\}, \{d\}, \{c, e\}\}.$$

It is easy to verify that $Q^3(\mathcal{F}_3)$.

4.5 Discussion

Conclusion. Our work shows how trust assumptions of (possibly disjoint) systems can be composed deterministically, allowing groups of strangers to join each other and collaborate under common trust assumption. We have presented three composition rules of increasing suitability. These rules are static and do not require interaction or agreement on the new trust assumption among the processes. The last one, the *Cartesian composition* rule, ensures that, if the original systems allow for running a particular protocol (guaranteeing consistency and availability), then so will the composed system. At the same time, the composed system tolerates as many faults as possible, subject to the underlying consistency and availability properties.

Extension to asymmetric Byzantine quorum systems. Recently, and particularly in the context of blockchains, even more general and flexible trust models have also been considered [107, 92, 112, 55, 37]. It is evident that a common trust model cannot be imposed in an open and decentralized or permissionless environment. Instead, every process should be free to choose who to trust. Cachin and Tackmann [37] introduce *asymmetric* Byzantine quorum systems. They let every process

specify their own fail-prone system and quorum system. Asymmetric Byzantine consensus protocols have been described as well [38]. Alpos, Cachin, and Zanolini [9] present composition rules for the asymmetric model. As there are no common trust assumptions in systems with asymmetric trust, the authors define the notion of a *tolerated system* as a representation of the failures the system can tolerate. Then they use the Cartesian composition, presented in this chapter, to compose asymmetric Byzantine quorum systems.

Chapter 5

Synchronization power of token smart contracts

Guerraoui *et al.* [86] prove that the *asset transfer* object, which allows assets to be moved across different processes, can be implemented without consensus. They also define the *k-asset transfer* object, where up k processes may share ownership of assets, and show that consensus is needed only among those k processes. The parameter k is fixed at creation time of the object. In this chapter we investigate the synchronization power (formally specified by the notion of the *consensus number*) of token smart contracts, focusing mainly on the ERC20 contract [160]. We prove that the consensus number changes dynamically according to the state of the object. To prove this result, we develop a dedicated methodology that accounts for the fact that certain method invocation may change the consensus number of an object.

Similar results are proved by subsequent work for more objects. Frey, Gestin, and Raynal [77] study the synchronization power of *AllowList* and *DenyList* objects, showing that no consensus is required to implement the former, while consensus among a specific set of processes is required for the latter. The authors show how these objects can be used to build an *anonymous asset transfer* protocol.

5.1 Introduction

The rise of cryptocurrencies has motivated the development of distributed applications running over blockchain platforms. These applications go far beyond the concept of a decentralized cryptocurrency, as initially envisioned by Bitcoin [124]. Taking this diversity to the extreme, *smart contracts* enable a blockchain to execute arbitrary programs, in a fully decentralized fashion akin to a "world computer." Introduced by Ethereum [71], smart contracts come in many different flavors and are the key element in most blockchain projects today.

Regardless of the type of supported smart contract, blockchain platforms rely on a distributed protocol that orders transactions and emulates a ledger data structure. A transaction may be a simple "coin transfer" in a cryptocurrency or a complex method call to a decentralized application. For either use-case, it is widely accepted that the blockchain nodes must execute all transactions in the same order to ensure consistency [154, 145]. That is, to ensure that the emulated ledger is consistent, transactions are sent using protocols that implement *total-order broadcast* or *consensus*. Garay *et al.* [78] showed such an equivalence formally for the Bitcoin protocol. This common theme seems to suggest that total order is also *necessary* for the consistency of blockchains.

However, this folklore intuition is wrong: consensus is not necessary to avoid double-spending in cryptocurrency applications. Guerraoui *et al.* [86] show this formally: after distilling the essence of a cryptocurrency protocol to the problem of realizing a consistent *asset transfer* (AT), the authors cast the latter as a sequential object in the shared-memory model and prove the AT object has *consensus number* 1 in the wait-free hierarchy [88]. In other words, consensus is not needed at all for emulating the functions of Bitcoin! The consensus number is a well-established tool to express the synchronization requirements of asynchronous concurrent objects. Informally, it provides an upper bound for the number of processes that can be synchronized using (arbitrarily many) instances of a given object. For cryptocurrencies modeled after Bitcoin that support shared accounts with up to k owners, Guerraoui *et al.* introduce a *k-shared* asset transfer (k-AT) object that has consensus number k, which is as powerful as consensus among its k owners. Going beyond their theoretical elegance, these results are of great practical interest because they pave the way to consensus-free

implementations of cryptocurrencies [50, 85], with higher efficiency and robustness to network partitions. In this particular case, for example, only the k owners need to reach consensus for spending from the account, provided they have additional means to publicize this widely in the network.

In this work, we investigate the synchronization power of smart contracts. We observe that although k-AT would allow to *generically* implement any smart contract among k processes, it remains open whether this level of synchronization is *necessary* for widely-used blockchain applications. We focus our attention on smart contracts for Ethereum, which is by far the most important platform for hosting decentralized applications. Moreover, many other networks have adopted its programming model. We present an abstraction of a token object that captures and generalizes the functionality of an ERC20 contract [160], which forms the basis for countless applications on Ethereum that hold billions today. Notice that the k-AT abstraction [86] applies to Bitcoin and its UTXO model of a currency. Ethereum, on the other hand, uses accounts, and ERC20 contracts are considerably more powerful than Bitcoin transactions. The additional features of ERC20 make it possible, for example, to let account owners *conditionally* issue transfers to other users of their choosing.

Empowering account owners to approve other spenders makes the ERC20 token object strictly more powerful than k-AT. In addition, approval of new spenders can be performed flexibly, at any time and for arbitrary amounts of tokens, achieving a dynamic that has no counterpart in the case of k-AT. Because of these differences, the results established for k-AT [86] cannot be lifted to ERC20 tokens. What crucially distinguishes an ERC20 token object from k-shared asset transfer is the increased level of dynamicity, which is reflected in its synchronization requirements. Namely, the consensus number of ERC20 tokens depends on the number of approved spenders for the same account, which may change as the account owner enables more spenders. Based on the observations, we develop a formalism to express that the consensus number of a token object can change over time, depending on the object's state. More concretely, we prove that there exist specific states from which it is possible to solve consensus among k processes, for every $k \leq n$ where n is the number of accounts defined by the token contract. Moreover, these states can be reached by letting any of the account owners approve new spenders.

Establishing the synchronization power of smart contracts is impor-

tant for understanding the level of synchronization that is required to run decentralized applications in a blockchain network. Not every two users must be synchronized on every aspect of their respective states, this only matters when their actions affect each other. Identifying the level of consensus needed for different applications also paves the way for realizing more efficient blockchain networks, which may exploit more parallelism.

Organization. After discussing related work in Section 5.2, in Section 5.3 we present relevant notations and background concepts. We then describe ERC20 tokens in Section 5.4, and we analyze their synchronization requirements in Section 5.5. In Section 5.6, we discuss other notable token standards and elaborate on extending our results to these tokens. Section 5.7 concludes our work suggesting future research directions.

5.2 Related work

Synchronization requirements and blockchain scalability. Blockchain technologies have pushed remarkable efforts towards designing more scalable platforms, fostering renovated interest in distributed consensus. Perhaps surprisingly, the idea of realizing a consensus-less decentralized cryptocurrency appeared only recently [87]. To formally prove that consensus is not necessary to realize a decentralized cryptocurrency, Guerraoui *et al.* [86] propose a shared-memory abstraction for the asset transfer problem as implemented in Bitcoin [124], and show that such abstraction requires only a minimal level of synchronization. Specifically, asset transfer has consensus number 1 in Herlihy's wait-free hierarchy [88]. The approach of analyzing the synchronization requirements of shared objects in terms of consensus number has been used by others. For instance, Cachin *et al.* [33] study the consensus number of various cloud-storage abstractions, and find that a key-value store has the weakest synchronization power (i.e., its consensus number is 1) while a replica object requires the strongest synchronization level (i.e., its consensus number is ∞).

Obviating the need to reach agreement on the exact ordering of transactions opens the door to more scalable solutions than the currently deployed, consensus-based blockchains. In this context, Collins *et al.* [50] present a decentralized payment system based on Byzantine reliable

broadcast. Guerraoui *et al.* [85] generalize the Byzantine reliable broadcast abstraction to the probabilistic setting and propose a protocol which efficiently realizes it, with the goal of replacing the usual quorum-based safety notions with stochastic guarantees for consistency in a distributed network. While the above-mentioned protocols fulfill the synchronization requirements for implementing asset transfer, and can therefore support plain cryptocurrency applications, they are not sufficient for the implementation of generic smart contracts.

Many other approaches have been explored in order to increase blockchain scalability [52], most prominently "on-chain" proposals such as optimized BFT-based consensus protocols [82, 30, 163], DAG-based protocols [93, 152], and sharding [164, 102, 162, 73], as well as "off-chain" solutions such as payment channels [134, 67] and sidechains [16]. Even though these alternative approaches have received a lot of attention recently [165], they have not yet been widely adopted in practice.

Smart contracts and Ethereum tokens. Ethereum [71] is the first open-source cryptocurrency platform supporting smart contracts, and it provides a decentralized virtual machine for executing arbitrary Turing-complete programs. The ERC20 standard, introduced by Buterin and Vogelsteller [160], provides functions for handling tokens over Ethereum, allowing users to transfer various types of transferable goods such as digital and physical assets. It formulates a common interface for fungible tokens and has become the most widely-deployed API for implementing a token functionality, with more than half of the overall Ethereum transactions being ERC20 token transfers [158].

5.3 Background

5.3.1 Shared memory objects

We begin with presenting well-established concepts from the concurrent computing literature. We mostly follow the standard notations and nomenclature [32, 140, 86].

Concurrent objects. We assume a (finite) set Π of processes that communicate in an asynchronous manner by invoking operations on, and receiving responses from, shared objects. Processes are sequential,

meaning that no process invokes a new operation before completing (i.e., receiving the response from) all previously invoked operations. We assume a *crash-failure* model: a process may halt prematurely, in which case we say the process has crashed. We say that a process is *faulty* if it crashes during its execution, otherwise we say that it is *correct*.

An *object type* (or simply *object*) defines the functionality of shared-memory programming abstractions providing a finite set of operations. We consider *concurrent* objects, namely objects which can be accessed by multiple processes simultaneously and concurrently. The specification of these objects can be sequential or not, where "sequential" means that all correct behaviors of the object can be described with sequences of invocations and responses (traces). In this work, we are only concerned with sequential objects. We define an object type as a tuple $T = (Q, q_0, O, R, \Delta)$, where Q is a set of states, $q_0 \in Q$ is an initial state, O is a set of operations, R is a set of responses, and $\Delta \subseteq Q \times \Pi \times O \times Q \times R$ defines the valid state transitions. We write $(q, p, o, q', r) \in \Delta$ to denote that process p invokes operation o on the object in current state q, and the operation completes by returning response r and causing the object to enter state q'.

An *implementation* for an object type T is a distributed algorithm describing, for each process, sufficient steps to realize each of the object's operations in such a way that desired safety and liveness properties are met. The strongest liveness condition for object implementations is *wait-freedom* [88], requiring that every invocation of any object operation terminates, despite process failures.

Registers. The simplest object type is a *register*, which defines a shared-memory functionality providing *read* and *write* operations. Given a register R, a process can write a value v into R by invoking $R.write(v)$; upon completion of this operation, the process is given TRUE in response. Similarly, a process can initiate a read operation on R by invoking $R.read()$; the process obtains a value R stores. In this work, we consider *atomic* registers. Formally, an atomic register provides *termination*, i.e., if a correct process invokes an operation, then the operation eventually completes, and *validity*, i.e., a read that is not concurrent with a write returns the last value written, while a read that is concurrent with a write returns the last value written or the value concurrently being written. Moreover, an atomic register provides *ordering*, i.e., if a read returns a value v and a subsequent read returns a value w, then

the write of w does not precede the write of v. This property implies that every operation of an atomic register can be thought to occur at a single indivisible point in time, which lies between the invocation and the completion of the operation [32].

Asset transfer (AT). In the context of analyzing blockchain applications from a concurrency-theory viewpoint, Guerraoui *et al.* [86] propose the asset transfer object as an abstraction for cryptocurrencies. We reproduce this abstraction below. Let $\mathcal{A}$ be a finite set of accounts, $|\mathcal{A}| = n$, and let $\mu\colon \mathcal{A} \to 2^{\Pi}$ denote the owner map that associates each account $a \in \mathcal{A}$ to the set of processes sharing the account. If $|\mu(a)| = k$, we say that a is a k-shared account.

Definition 9 (Asset transfer). The *asset transfer* object associated to $\mathcal{A}$ and μ, denoted by $AT = (Q, q_0, O, R, \Delta)$, is defined as follows:

- Set Q contains all *balance* maps, i.e., $Q = \{\beta\colon \mathcal{A} \to \mathbb{N}\}$.
- The initialization map $q_0 = \beta_0$ assigns an initial balance to each account.
- O contains two operations, $O = \{\mathit{transfer}(a_s, a_d, v) : a_s, a_d \in \mathcal{A}, v \in \mathbb{N}\} \cup \{\mathit{balanceOf}(a) : a \in \mathcal{A}\}$, where $\mathit{transfer}(a_s, a_d, v)$ lets the caller process, say p, transfer v tokens from a source account a_s to a destination account a_d, provided that $p \in \mu(a_s)$, and $\mathit{balanceOf}(a)$ reads the balance of account a.
- R contains the possible responses to operations in O, $R = \{\text{TRUE}, \text{FALSE}\} \cup \mathbb{N}$.
- Δ defines the valid state transitions. Given a state $q = \beta \in Q$, a process $p \in \Pi$ with account a_p, an operation $o \in O$, a response $r \in R$, and a new state $q' = \beta' \in Q$, we have $(q, p, o, r, q') \in \Delta$ if and only if either of the following conditions holds:
  - $o = \mathit{transfer}(a_s, a_d, v) \ \wedge \ p \in \mu(a_s) \ \wedge \ \beta(a_s) \geq v \ \wedge \ \beta'(a_s) = \beta(a_s) - v \ \wedge \ \beta'(a_d) = \beta(a_d) + v \ \wedge \ \forall c \in \mathcal{A} \setminus \{a_s, a_d\} : \beta'(c) = \beta(c) \ \wedge \ r = \text{TRUE}$;
  - $o = \mathit{transfer}(a_s, a_d, v) \ \wedge \ (\beta(a_s) < v \ \vee \ p \notin \mu(a_s)) \ \wedge \ q' = q \ \wedge \ r = \text{FALSE}$;
  - $o = \mathit{balanceOf}(a) \ \wedge \ q' = q \ \wedge \ r = \beta(a)$.

If the maximum number of processes sharing an account is k, we name the object a k-shared asset transfer, denoted by k-AT.

5.3.2 Synchronization power

Consensus. Another important object type is *consensus*, which allows a set of processes to agree on a value. A consensus object C provides a single operation *propose*. A process can invoke $C.propose(v)$ on input a proposal v as a candidate value to be agreed upon. Every process can call *propose* with their own proposed value, and only one invocation is permitted (i.e., it is a "single-shot" object). Upon completion, the operation returns a value d, called the decided value. Besides wait-freedom (a.k.a. *termination*), we require *validity*, i.e., the decided value is the proposal v of some process, and *consistency*, i.e., every process returns the same decided value d.

Synchronization power of shared objects. The prominent result by Fischer, Lynch, and Paterson [76] establishes the impossibility of wait-free implementing consensus from atomic registers. This means that consensus requires a higher level of synchronization than atomic registers. In fact, the consensus object is *universal*, in the sense that any shared object described by a sequential specification can be wait-free implemented from consensus objects and atomic registers [88]. Therefore, consensus can be used to reason about the synchronization power of all shared objects (which admit a sequential specification) among a number of processes. This leads to the central concept of *consensus number* to express the synchronization power of shared objects.

Definition 10 (Consensus number [88]). The *consensus number* associated with an object O is the largest number n such that it is possible to wait-free implement a consensus object from atomic registers and objects of type O. If there is no largest n, the consensus number is said to be infinite. Given an object O, we denote its consensus number by $\mathcal{CN}(O)$.

The consensus number allows comparing objects based on their synchronization power, thereby establishing a hierarchy among objects—the consensus hierarchy. In this work, we leverage the concept of consensus number to study the level of synchronization required for popular smart-contracts tokens.

Theorem 16 ([88]). *Let O and O' be two objects such that $\mathcal{CN}(O) = n$ and $\mathcal{CN}(O') > \mathcal{CN}(O)$. Then there is no wait-free implementation of an object of type O' from objects of type O and read/write registers in a system of n processes.*

5.4 Defining ERC20 tokens as shared objects

In this section, we present a smart contract for transferring tokens according to the Ethereum Request for Comment (ERC) 20 specification and propose a corresponding shared-memory abstraction.

Tokens are blockchain-based assets which can be exchanged across users of a blockchain platform. Ethereum Request for Comment (ERC) 20 defines a standard for the creation of a specific type, dubbed ERC20 token, one of the most widely adopted tokens on Ethereum. ERC20 tokens are transferred through dedicated transactions among Ethereum addresses, and are managed by smart contracts. For completeness, we reproduce in Algorithm 7 the algorithmic specification defined in the ERC20 proposal [160], with minimal notational changes to ease the comparison with the objects defined in this work.

The definition of a token object we propose is a generalization of an ERC20 token. The reason for slightly deviating from the original specification (as per Algorithm 7) is that it represents a more expressive abstraction: it allows us to reason about synchronization requirements of ERC20 tokens as well as comparing them with the asset transfer object.

Let $\mathcal{A}$ be a finite set of accounts. We assume one account per process, $|\Pi| = |\mathcal{A}| = n$, and define a bijection $\omega \colon \mathcal{A} \to \Pi$ between accounts and processes, i.e., $\omega(a_i) = p_i$ for all $i \in \{1, \ldots, n\}$. We name $\omega \colon \mathcal{A} \to \Pi$ the *owner map* that associates to each account a the corresponding process $\omega(a)$ which owns the account.[1] To simplify the notation, we use the shorthand a_p for the account owned by process p, i.e., such that $\omega(a_p) = p$.

Notice that in the case of asset transfer (cf. Definition 9), *account ownership* captures a slightly different setting than compared to token objects: the former allows for shared ownership while the latter does not. We make this explicit by using different owner maps μ and ω, respectively. However, as we explain in detail in the next section, ERC20 tokens offer a richer set of operations that, among others, enables a conditioned form of shared ownership.

[1] Although we use a similar formalism as Guerraoui *et al.* [86] to define account ownership, we make the restriction to single-owner accounts to meet the Ethereum-token specification. As we will see in later sections, in Ethereum tokens there are no shared accounts, however, a similar concept is enabled by means of dedicated methods.

Algorithm 7 Sequential specification of ERC20 functionalities. Code for process p_i.

State:

115: *const d*, the process that deployed the contract
116: *const string name*
117: *const string symbol*
118: *const int decimals*
119: *const int totalSupply*
120: *balances* $[] \subseteq \mathcal{A} \times \mathbb{N}$, init. $balances[d] = totalSupply, balances[i] = 0, \forall i \neq d$
121: *allowances* $[][] \subseteq \mathcal{A} \times \mathcal{P} \times \mathbb{N}$, initially $\emptyset$

122:**operation** *totalSupply*()
123: **return** *totalSupply*

124:**operation** *balanceOf*(*owner*)
125: **return** *balances*[*owner*]

126:**operation** *transfer*(*to*, *value*)
127: **if** $balances[p_i] < value$ **then**
128: **return** FALSE
129: **else**
130: $balances[p_i] \leftarrow balances[p_i] - value$
131: $balances[to] \leftarrow balances[to] + value$
132: **return** TRUE

133:**operation** *transferFrom*(*from*, *to*, *value*)
134: **if** $allowances[from][p_i] < value$ **then**
135: **return** FALSE
136: **else if** $balances[from] < value$ **then**
137: **return** FALSE
138: **else**
139: $allowances[from][p_i] \leftarrow allowances[from][p_i] - value$
140: $balances[from] \leftarrow balances[from] - value$
141: $balances[to] \leftarrow balances[to] + value$
142: **return** TRUE

143:**operation** *approve*(*spender*, *value*)
144: $allowances[p_i][spender] \leftarrow value$
145: **return** TRUE

146:**operation** *allowance*(*owner*, *spender*)
147: **return** *allowances*[*owner*][*spender*]

Using this notation, below we provide a specification for the *ERC20 token* smart contract using the formalism of shared objects (cf. Section 5.3.1).

Definition 11 (ERC20 token object). Let $\mathcal{A}$ be a set of accounts and let Π be the set of corresponding owner processes. A *token object* T associated to $\mathcal{A}$ consists of a tuple $T = (Q, q_0, O, R, \Delta)$, where:

States: Q contains all *balances* and *allowances* maps, i.e.,

$$Q = \{\beta \colon \mathcal{A} \to \mathbb{N}\} \times \{\alpha \colon \mathcal{A} \times \Pi \to \mathbb{N}\}. \tag{5.1}$$

Intuitively, for $a \in \mathcal{A}$ and $p \in \Pi$, $\beta(a)$ indicates the balance of account a, and $\alpha(a, p)$ denotes the amount of tokens that process p is allowed to spend from account a.

Initial state: $q_0 = (\beta_0, \alpha_0)$ denotes the pair of initial account balances and allowances.

Operations: O contains the following operations:

$$\begin{aligned} O = &\{\mathit{transfer}(a_d, v) : a_d \in \mathcal{A}, v \in \mathbb{N}\} && (5.2)\\ &\cup \{\mathit{transferFrom}(a_s, a_d, v) : a_s, a_d \in \mathcal{A}, v \in \mathbb{N}\} && (5.3)\\ &\cup \{\mathit{approve}(p, v) : p \in \Pi, v \in \mathbb{N}\} && (5.4)\\ &\cup \{\mathit{balanceOf}(a) : a \in \mathcal{A}\} && (5.5)\\ &\cup \{\mathit{allowances}(a, p) : a \in \mathcal{A}, p \in \Pi\}. && (5.6) \end{aligned}$$

Operation $\mathit{transfer}(a_d, v)$ lets the caller process, say p, transfer v tokens from its account a_p to destination account a_d; similarly, $\mathit{transferFrom}(a_s, a_d, v)$ lets the caller process transfer v tokens from source account a_s to destination account a_d. Operation $\mathit{approve}(p', v)$ lets the caller process p authorize another process p' to transfer up to v tokens from p's account. Finally, $\mathit{balanceOf}(a)$ reads the balance of account a, and $\mathit{allowances}(a, p)$ reads the amount of tokens that process p is authorized to transfer from a.

Responses: R contains the possible responses for all operations in O, namely $R = \{\text{TRUE}, \text{FALSE}\} \cup \mathbb{N}$.

Sequential specification: Δ defines the valid state transitions. Given a state $q = (\beta, \alpha) \in Q$, a process $p \in \Pi$ with account a_p, an operation $o \in O$, a response $r \in R$, and a new state $q' = (\beta', \alpha') \in Q$, we have $(q, p, o, r, q') \in \Delta$ if and only if either of the following conditions holds:

- $o = \mathit{transfer}(a_d, v) \ \wedge \ \beta(a_p) \geq v \ \wedge \ \beta'(a_p) = \beta(a_p) - v \ \wedge \ \beta'(a_d) = \beta(a_d) + v \ \wedge \ \forall c \in \mathcal{A} \setminus \{a_p, a_d\} : \beta'(c) = \beta(c) \ \wedge \ \alpha' \equiv \alpha \ \wedge \ r = \text{TRUE}$;
- $o = \mathit{transfer}(a_d, v) \ \wedge \ \beta(a_p) < v \ \wedge \ q' = q \ \wedge \ r = \text{FALSE}$.
- $o = \mathit{approve}(\bar{p}, v) \ \wedge \ \alpha'(a_p, \bar{p}) = v \ \wedge \ \alpha'(a, p) = \alpha(a, p) \ \forall (a, p) \neq (a_p, \bar{p}) \ \wedge \ \beta' \equiv \beta \ \wedge \ r = \text{TRUE}$;
- $o = \mathit{transferFrom}(a_s, a_d, v) \ \wedge \ \beta(a_s) \geq v \ \wedge \ \alpha(a_s, p) \geq v \ \wedge \ \beta'(a_s) = \beta(a_s) - v \ \wedge \ \alpha'(a_s, p) = \alpha(a_s, p) - v \ \wedge \ \beta'(a_d) = \beta(a_d) + v \ \wedge \ \alpha'(a, p) = \alpha(a, p) \ \forall (a, p) \neq (a_s, p) \ \wedge \ \forall c \in \mathcal{A} \setminus \{a_s, a_d\} : \beta'(c) = \beta(c) \ \wedge \ r = \text{TRUE}$;
- $o = \mathit{transferFrom}(a_s, a_d, v) \ \wedge \ (\beta(a_s) < v \vee \alpha(a_s, p) < v) \ \wedge \ q' = q \ \wedge \ r = \text{FALSE}$;
- $o = \mathit{balanceOf}(a) \ \wedge \ q = q' \ \wedge \ r = \beta(a)$;
- $o = \mathit{totalSupply} \ \wedge \ q = q' \ \wedge \ r = \sum_{a \in \mathcal{A}} \beta(a)$;
- $o = \mathit{allowances}(a, \bar{p}) \ \wedge \ q' = q \ \wedge \ r = \alpha(a, \bar{p})$.

In this chapter we use of the following shortcut notation. For every state $q \in Q$, we write T_q to denote the token object initialized with state q, i.e., $T = (Q, q, O, R, \Delta)$. Similarly, for $Q' \subseteq Q$ we write $T_{Q'}$ to indicate that token object is initialized with any state $q \in Q'$. Finally, we note that in the ERC20 standard [160] (cf. Algorithm 7) the state of the smart contract is fully specified by the arrays *balances*[ ] and *allowances*[ ]. Namely, for all $a \in \mathcal{A}$ and all $p \in \Pi$, we have $T.\mathit{balances}[a] = T.\beta(a)$ and $T.\alpha(a, p) = T.\mathit{allowances}[a][p]$.

The example below illustrates the various ERC20 token operations and their interplay.

Example 5 (ERC20 token: sample execution). Consider a set of three processes, $\Pi = \{A, B, C\}$ (Alice, Bob, and Charlie), and corresponding accounts, $\mathcal{A} = \{a_A, a_B, a_C\}$. Let Alice be the deployer of an ERC20 token contract, and suppose Alice provides an initial supply of 10 tokens, i.e., $\mathit{totalSupply} = 10$. According to the ERC20 specification, the token object T associated to the contract is initialized in a state q_0, where $\mathit{balances}[a_A, a_B, a_C] = [10, 0, 0]$ and $\forall a \in \mathcal{A} : \mathit{allowances}[a][A, B, C] = [0, 0, 0]$. Starting from this initial configuration, let Alice invoke $\mathit{transfer}(a_B, 3)$, sending 3 tokens to Bob's account. This operation triggers the transfer of 3 tokens from account a_A to account a_B and, upon completion, it causes an update to state q_1, where $\mathit{balances}[a_A, a_B, a_C] = [7, 3, 0]$. Let now

Bob invoke *approve*$(C, 5)$, authorizing Charlie to transfer up to 5 tokens from account a_B. Upon completion, this operation causes an update to state q_2, with $\mathit{allowances}[a_B] = [0, 0, 5]$. Upon being approved, let Charlie invoke *transferFrom*$(a_B, a_C, 5)$ to transfer 5 tokens from Bob's account to his own account. Despite the fact that Charlie's allowance $\mathit{allowances}[a_B][C] = 5$ would in principle permit such transfer, Bob's balance $\mathit{balances}[a_B] = 3$ is currently insufficient. Therefore, the operation returns FALSE, leaving the state unmodified, i.e., $q_3 = q_2$. Finally, let Charlie invoke *transferFrom*$(a_B, a_A, 1)$ to transfer 1 token from account a_B to Alice's account. This time, the amount of tokens to be transferred is below the account balance and, upon completion, the operation triggers an update to state q_4, where $\mathit{balances}[a_A, a_B, a_C] = [8, 2, 0]$ and $\mathit{allowances}[a_B] = [0, 0, 4]$.

5.5 Consensus number of ERC20 tokens

In this section, we study the synchronization power of the ERC20 token object by analyzing its consensus number.

5.5.1 Overview of the results

The consensus number of an ERC20 token object can be expressed in terms of the maximum number of processes that can transfer tokens from the same account. This number, denoted below by k, depends on the account balances and allowances defined by the object's state $q = (\beta, \alpha)$, and hence it can change as the state is updated. In the rest of this section, we therefore analyze the consensus number of a token object T for various state configurations.

Approach and challenges. Given the similarities between the ERC20 token object and the k-shared asset transfer object, one may think they have the same consensus number. Intuitively, the *approve* method in ERC20 tokens allows emulating shared accounts by letting every account owner authorize other processes to transfer tokens from its own account. In fact, there are at least two peculiarities of ERC20 tokens which depart from k-shared accounts. Firstly, a k-shared asset transfer supports at most k owners per account, where $k \leq n$ is fixed upfront (because the owner map μ in k-AT is *static*). This is in contrast with ERC20 tokens, where each account owner can *dynamically* add and

remove spenders at any time of the execution, and the number of valid spenders per account is subject to change as the protocol is ongoing. In other words, an ERC20 token object could be loosely seen as a k-shared asset transfer with k changing dynamically. Secondly, in k-AT the owners of a shared account remain owners for the whole lifetime of the object, i.e., they all can transfer tokens from that account as long as the balance is positive. In ERC20 tokens instead, an approved spender remains a valid spender until it consumes the granted allowance or the account owner decides to revoke the spender's allowance (this can be done by resetting the allowance to the default value 0).

These crucial differences show a separation between the k-AT object and the ERC20 token object, and suggest that the two objects meet different synchronization requirements. In particular, it is not possible to apply known results and techniques for k-AT to the case of ERC20 tokens. Moreover, the approval mechanism to add and remove spenders in ERC20 tokens has subtle implications on the object's synchronization power.

In the rest of this section, we confirm these observations formally and make precise statements about the consensus number of the ERC20 token object. We now provide a rather informal summary of our results, which we state in full detail and prove in Section 5.5.2. The statements below hold for every $k \leq n$.

Lower bound. There exists a set S_k of states, which we name *synchronization states*, such that for every $q \in S_k$ it is possible to wait-free implement a consensus object among k processes using objects of type T_q (Theorem 17). Formally:

$$\mathcal{CN}(T_{S_k}) \geq k. \tag{5.7}$$

To prove this lower bound, we show that a consensus object supporting k processes reduces to T_q, with $q \in S_k$, by presenting a wait-free implementation of consensus for k processes from objects of type T_q and atomic registers.

Upper bound. The set of states can be partitioned into $[Q_1, \ldots, Q_n]$, with $Q = \cup_{k=1}^{n} Q_k$, so that for every $q \in Q_k$, at most k processes can reach consensus using token objects of type T_q (Theorem 18). Formally:

$$\mathcal{CN}(T_{Q_k}) \leq k. \tag{5.8}$$

Proving the upper bound turns out to be more involved. We proceed with an indirect argument, showing that the hypothesis $\mathcal{CN}(T_{Q_k}) = k' > k$ leads to a contradiction. Intuitively, the contradiction is reached because no implementation of consensus for k' processes from T_q, with $q \in Q_k$, can be wait-free.

5.5.2 Technical results and proofs

Essentially, we show that an ERC20 token represents a dynamic k-shared AT object, where k depends on the current object's state. Specifically, k is the maximum number of valid spenders for the same account. For each $k \leq n$, where n is the total number of accounts, there exists a class S_k of states, the class of *k-synchronization states*, such that $\forall q \in S_k$, it holds $\mathcal{CN}(T_q) \geq k$. However, we cannot conclude that $\mathcal{CN}(T) = \infty$. We can only say that if a state $q \in S_n$ is reached, then we can solve consensus among all processes. That is, there exists a state $q \in S_n \subset Q$ such that $\mathcal{CN}(T_q) = n$. This is weaker than saying that for every state, we can solve consensus among n processes. In particular, it is not possible to reach such a state $q \in S_n$ in a wait-free manner, as we prove later.

Let us first define the sets S_k of synchronization states formally, then we will provide relevant bounds for the consensus number of an ERC20 token object in a synchronization state.

Enabled spenders. For every state $q = (\beta, \alpha) \in Q$, let $\sigma_q \colon \mathcal{A} \to 2^{\Pi}$ denote the mapping associating each account a to its *enabled spenders* according to q, i.e., the set of processes that are enabled to transfer tokens from account a w.r.t. balances β and allowances α specified by state q. Formally,

$$\sigma_q(a) = \{p \in \Pi : p = \omega(a) \ \vee \ \alpha(a, p) > 0\}. \tag{5.9}$$

Note that we explicitly include the account owner $\omega(a)$ in the set of enabled spenders for account a. We conventionally assume that an account with zero balance has only its owner as enabled spender, i.e., $\beta(a) = 0 \implies \sigma_q(a) = \{\omega(a)\}$. Indeed, even if there may be some process p, other than the owner, with positive allowance for account a, i.e., $\beta(a) = 0$ and $\alpha(a, p) > 0$, this process would not be able to transfer tokens from a unless the balance is increased.

State partition. Let Q_k, with $k \leq n$, be the set of states with exactly k valid spenders from the same account, i.e.,

$$Q_k = \{q \in Q : \max_{a \in \mathcal{A}} |\sigma_q(a)| = k\}. \tag{5.10}$$

Observe that the subsets $Q_1, \ldots, Q_n$ define a partition. Intuitively, we would like to say that each subset is associated to a given level of synchronization, defining a hierarchy $Q_1 \prec \cdots \prec Q_n$ reflecting the synchronization level, where Q_k corresponds to consensus number k. Importantly, the level of synchronization may change as the object's state is updated. In fact, for every k and for all states $q \in Q_k$, there exists a valid transition $(q, p, o, r, q') \in \Delta$, such that

$$q \in Q_k, \quad p = \omega(a), \quad o = \textit{approve}, \quad r = \text{TRUE}, \quad q' \in Q_{k+1}, \tag{5.11}$$

in the sense that it is possible to reach some state in Q_{k+1} from $q \in Q_k$. However, *the only way to do so* is by letting *the owner* of a k-spender account a approve a new spender.

Synchronization states. Later in this section, we show how to implement consensus from an ERC20 token object. Intuitively, we leverage an account for which multiple spenders have been approved: we let the spenders engage in a "race" where they compete for spending the account's tokens, and the "winner" of this competition gets to choose the decided value (in the consensus protocol). This idea crucially relies on the fact that there is a *unique* winner. To guarantee this, we need to impose an additional requirement on the balance and allowances of the account used in the implementation. We formally specify such requirement by defining predicate $\mathtt{U}\colon \mathcal{A} \times Q \to \{\text{TRUE}, \text{FALSE}\}$ (ensuring unique transfers) as follows. Namely, given a state $q = (\beta, \alpha)$ and an account a, we define:

$$\begin{aligned} &\mathtt{U}(a, q) \quad \text{if and only if} \quad \beta(a) > 0 \quad \wedge \\ &(|\sigma_q(a)| \leq 2 \quad \vee \quad \forall p_i, p_j \in \sigma_q(a) \setminus \{\omega(a)\} \ : \ \alpha(a, p_i) + \alpha(a, p_j) > \beta(a)) \,. \end{aligned} \tag{5.12}$$

We introduce further notation to identify relevant states which will appear in our main results. For every k as above, we define $S_k \subset Q_k$ to be the set of states q with exactly k valid spenders from the same account a and such that predicate $\mathtt{U}$ holds for (a, q):

$$S_k = \{q \in Q \ : \ \exists a \in \mathcal{A} \ : \ |\sigma_q(a)| = k \ \wedge \ \mathtt{U}(a, q)\} \tag{5.13}$$

We refer to the states in S_k as *k-synchronization states*. Intuitively, $q \in S_k$ are the states from which we can solve consensus for k processes, i.e., using an object type T_q, but not for more than k processes.

Theorem 17. *For every $k \leq n$ it holds $\mathcal{CN}(T_{S_k}) \geq k$.*

Proof. We show an implementation of a consensus object C for k processes, using an instance of a T_q object, with $q \in S_k$, and k atomic registers $R[1], \ldots, R[k]$. By the hypothesis $q \in S_k$, at least one account has k enabled spenders (cf. (5.13)) and satisfies the requirements defined by predicate `U` (defined in (5.12)) with respect to state $q = (\beta, \alpha)$. Without loss of generality, let $a_1 \in \mathcal{A}$ denote one such account, and let $\sigma_q(a_1) = \{p_1, \ldots, p_k\}$ with $p_1 = \omega(a_1)$. Let $B = \beta(a_1)$ and $A_j = \alpha(a_1, p_j)$, $j \in \{2, \ldots, k\}$, denote the balance of a_1, resp., the allowances of processes $p_2, \ldots, p_k$ for account a_1, w.r.t. state $q = (\beta, \alpha)$. Let a_d be any account in $\{a_2, \ldots, a_k\}$. We assume the processes know the designated account a_d prior to executing the protocol. The code for the implementation is shown in Algorithm 8, and described below.

Algorithm 8 Wait-free implementation of a consensus object C among k processes in $\{p_1, \ldots, p_k\}$ using an ERC20 object T_q, with $q \in S_k$, associated to an account set $\mathcal{A} = \{a_1, \ldots, a_n\}$, where $a_d \in \mathcal{A} \setminus \{a_1\}$. Code for process p_i.

```
    State
148:    R[j] ← ⊥, j ∈ {1, ..., k}
149:    An ERC20 object T initialized such that:
150:        T.balances[a_1] = B
151:        T.allowances[a_1][p_j] = A_j, j ∈ {2, ..., k}

152:operation propose(v)
153:    R[i].write(v)
154:    if p_i = p_1 then
155:        T.transfer(a_d, B)                    // Transfer full balance
156:    else T.transferFrom(a_1, a_d, A_i)
157:    for j ∈ {2, ..., k} do
158:        if T.allowances(a_1, p_j) = 0 then
159:            return R[j].read()
160:    return R[1].read()
```

Briefly, each process p_i writes its proposed value v in a register $R[i]$. Then process p_1 attempts to transfer its whole balance B to account

a_d, and process $p_i \neq p_1$ invokes operation T.*transferFrom* as an attempt to transfer its whole allowance A_i from a_1 to a_d. Since only one of the *transfer* and *transferFrom* invocations succeeds (as we prove shortly), we can safely decide the value proposed by the process which triggered the successful transfer. The intuition is that only the invocation of *transfer* by p_1 or the first completing invocation of *transferFrom* by some process p_{i^*}, for $i^* \in \{2, \ldots, k\}$, succeeds. Upon completion of that operation, no other process will be able to issue its own transfer because the balance of a_1 will be too low (this is guaranteed by the predicate `U` defined in (5.12)). Moreover, while the allowance of process p_{i^*} will be 0, the rest of the processes will still have positive allowances. Since the allowances can be read by all processes, every process can determine who won the competition and decide the value proposed by the winner. Therefore, once an operation *propose* completes by returning decision value v^*, every other process that invokes *propose* also decides v^*. More precisely, we select the "winner" process p_{i^*} as the one which succeeds in spending its allowance by transferring A_{i^*} tokens from a_1 to a_d. If none of the processes is found to have zero allowance, then p_1 must have been the first that called *propose*, and thus consumed the whole balance and caused any other calls to *propose* to fail.

We now show that the proposed implementation satisfies the *termination*, *validity*, and *agreement* properties of a consensus object (cf. Section 5.3). Regarding the termination property, observe that all instructions of operation C.*propose* do terminate: writing the proposed value to $R[i]$ terminates because of the use of an atomic register; the call to *transferFrom* terminates because it only involves reading from and writing to registers; the *for* loop is bounded by the number of processes k, and each iteration involves reading the allowance of a process p_j and potentially reading from the corresponding register $R[j]$ (termination follows by the properties of register R). The validity property holds because the decided value is read from one of the registers $R[i]$ written by process p_i, for $i \in \{1, \ldots, k\}$, and the proposal of each process p_i is written before the *read* operation on that register is invoked (this is enforced by the *if* condition, cf. line 158). Hence, the decided value must be the proposal of some process p_j, for $j \in \{1, \ldots, k\}$. As for the consistency property, as we already mentioned, only the first invocation of operation *transfer* or *transferFrom* may succeed. In the former case, no invocation to *transferFrom* can ever succeed, hence no allowance can be set to 0, hence all processes will return the value proposed by p_1. In the latter case, the allowance of one of the processes p_{i^*}, for $i^* \in \{2, \ldots, k\}$

will be decreased from A_{i^*} to 0, and the *if* condition (cf. line 158) ensures that only the register written by a process with an allowance of 0 may be read. □

The previous theorem provides a lower bound for the consensus number of a token object T_q with initial state $q \in S_k$. Therefore, so far we can deduce the following inequalities (where the right-most inequality trivially holds):

$$k \overset{\text{(Thm. 17)}}{\leq} \mathcal{CN}(T_{S_k}) \leq \infty \tag{5.14}$$

The upper bound in (5.14) is a loose one. We proceed with establishing a tight upper bound for the consensus number of T. Similarly to the case of the lower bound, we will need to condition our statement on the object's state.

Observe that starting from the initial state q_0 as defined in the original ERC20 specification—i.e., no process is authorized to issue transfers from accounts they do not own, and all but the contract deployer have zero balances (cf. Algorithm 7)—it is possible to reach a state $q \in S_k$ *as long as* tokens are transferred across accounts, and the owner of an account a with positive balance approves other $k-1$ spenders with sufficient allowances. Therefore, reaching a state in S_k is conditioned on all these $k-1$ *approve* operations succeeding, and ultimately on the account owner p_a not failing until then. Due to the above condition, a *wait-free* implementation of consensus from T_{q_0} is unachievable. More generally, starting from any state $q \in Q_k$, it is not possible to wait-free implement consensus among $k' > k$ processes, as we prove in the following theorem.

Theorem 18. *For every $k \leq n$ it holds $\mathcal{CN}(T_{Q_k}) \leq k$.*

Proof. We proceed by contradiction and assume a wait-free implementation of consensus for k' processes using objects of type T_{Q_k} and atomic registers, where $k' > k$, hence we show that for any such implementation there exists an infinite sequential execution that leaves it in a bivalent state.

Let us first recall some relevant terminology. A protocol state is *bivalent* if, starting from that state, there exists some execution in which the processes decide 0 and some execution in which they decide 1. A protocol state is called *critical* if it is bivalent and any subsequent state, reached by having a process invoke any of the object's methods, is univalent. Every wait-free consensus protocol has a critical state [88]. In the following, we denote one such state by q_c. Further, the invocation

which brings the protocol from a critical state to a univalent state is called a *decision step*.

Without loss of generality, let $p_1, p_2 \in \Pi$ be processes such that the decision step for p_1, denoted by o_1, brings the protocol into a 0-valent state, and the decision step for p_2, denoted by o_2, brings it into a 1-valent state. The rest of the proof is case analysis of the methods which p_1 and p_2 execute in these decision steps.

Let us first assume that the decision step for p_1 is to invoke any operation on an atomic register, while the decision step for p_2 is to invoke any operation on a T_{Q_k} object. Starting from q_c, the sequential execution of o_1 followed by o_2 brings the protocol into a 0-valent state q_1, since p_1 took a step first. Instead, the sequential execution of o_2 followed by o_1 brings the protocol into a 1-valent state q_2, since p_2 took a step first. However, the states q_1 and q_2 are identical, because the two operations o_1 and o_2 commute, a contradiction.

Let us now assume that at least one of the invocations, say o_1, is on a read-only method. Consider the sequential execution starting from q_c, where p_1 executes o_1, then p_2 executes o_2, resulting in state q_1, and then p_2 runs alone and terminates. In this execution, p_2 must decide 0, because p_1 took a step first. Consider now the execution starting from q_c, where p_2 executes o_2, resulting in state q_2, and then p_2 runs alone and terminates. In this execution, p_2 decides 1. However, the states q_1 and q_2 differ only in the internal values of p_1, since the latter invoked a read-only method, hence they are indistinguishable for p_2. Yet, p_1 decides a different value starting from q_1, respectively, q_2, a contradiction.

According to the commutativity and read-only arguments just described, the decision steps of p_1 and p_2 must operate on the same object and invoke a method that modifies the state of that object [88]. In the following, we examine all possible combinations for the decision steps, and whenever they commute, or are read-only, we refer to the arguments above to imply a contradiction.

Observe that the methods *totalSupply*, *balanceOf*, and *allowance* of the ERC20 token object are read-only, hence we do not examine them further. Moreover, if both o_1 and o_2 are *approve* invocations, or if one of them is an *approve* invocation and the other is a *transfer* invocation, then o_1 and o_2 commute and a contradiction is reached as shown above. We proceed by analyzing the remaining, non-trivial cases.

Case 1: both o_1 and o_2 are invocations to the transfer method. Since *transfer* withdraws tokens from the account of the calling process, o_1 and o_2 commute except for the case when $o_1 = \mathit{transfer}(a_2, x)$, that is,

a transfer of x tokens to the account of p_2, and the balance of p_2 is not sufficient to execute the transfer o_2 before o_1, that is, o_2 returns FALSE if executed before o_1. Observe that in this case, o_2 is equivalent to a read-only operation, therefore a contradiction is reached as described earlier. (For instance, consider the following two executions: in the first one, p_2 executes o_1 and then runs alone, deciding 0; in the second one, operation o_2 is executed first, followed by o_1, hence p_1 runs alone and decides 1.)

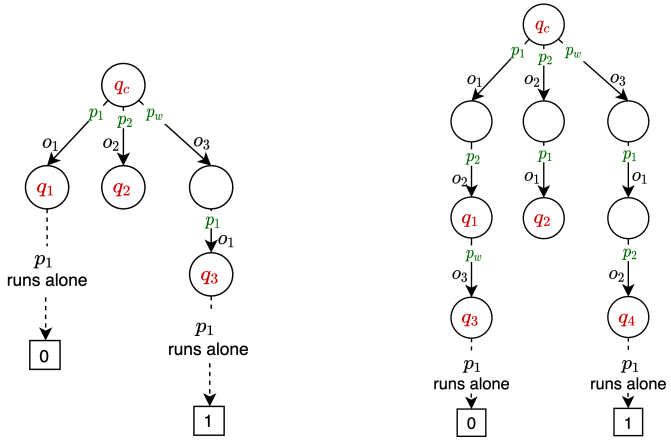
Case 2: both o_1 and o_2 are invocations to the transferFrom method. These invocations commute, except for the case when they both use the same source account a_s and the balance of a_s is only sufficient for one of the two transfers, and both processes are enabled to spend from a_s (without the latter condition the invocation would be equivalent to a read-only operation). Let us focus on this case. Since our implementation solves consensus among k' processes, and at most k processes are enabled spenders for the same account, where $k' > k$, there must be (at least) a process p_w that is not an enabled spender for account a_s—and by definition, p_w cannot be process $p_s = \omega(a_s)$. Assume wlog that the decision step o_3 taken by p_w brings the protocol in a 1-valent state (otherwise swap p_1 for p_2 in the following argument). Under this configuration, we will reach a contradiction for any possible method involved in o_3.

Let us begin with the case where o_3 is a *transferFrom* invocation with a_s as source account, as shown in Figure 5.1a.

As process p_w is not enabled for account a_s, operation o_3 returns FALSE without modifying the state, thus it is equivalent to a read-only operation. Let us now consider the following two executions: process p_1 executes o_1, reaching state q_1, and then runs alone, thus deciding 0; process p_w executes o_3, then process p_1 executes o_1 reaching state q_3, then process p_1 runs alone, thus deciding 1. We have a contradiction, because states q_1 and q_3 are indistinguishable to process p_1.

Moreover, if operation o_3 is a *transferFrom* invocation with source account a_t, with $t \neq s$, then operations o_1 and o_3 commute, and a contradiction is reached with a similar argument as above. A similar argument can be applied to all other possible methods, by observing that o_3 is either read-only (*totalSupply*, *balanceOf*, *allowance*), or it commutes with o_1 (*approve*, *transfer*), because $p_w \neq p_s$.

Case 3: operations o_1 and o_2 are a transfer, respectively, a transferFrom invocation, or vice versa. This case is analogous to the previous one. In-



(a) Case 2: both operations o_1 and o_2 are invocations to the *transferFrom* method.

(b) Case 4: operation o_1 is an *approve* invocation and o_2 is a *transferFrom* invocation.

Figure 5.1. Possible state transitions from the critical state q_c.

deed, if the *transferFrom* invocation has a source account other than a_1, then the two invocations commute, while if it has a_1 as source account, the same reasoning as in the previous case, making use of process p_w, applies.

Case 4: operation o_1 is an approve invocation and o_2 is a transferFrom invocation. Let us examine the case where o_1 approves process p_2 and o_2 uses a_1 as source account—in all other cases, the two invocations commute. We distinguish two cases.

In the first case, assume that p_2 is not already an enabled spender for account a_1. Then operation o_2, if executed before o_1, returns FALSE and hence it does not affect the state of T_{Q_k}. Therefore, o_2 is equivalent to a read-only operation and a contradiction is reached with the exact same executions as in Case 1 (see above).

In the second case, assume that p_2 is already an enabled spender for account a_1. Then, as depicted in Figure 5.1b, the states q_1 and q_2, reached by the sequential execution of o_1 and then o_2, respectively, by the sequential execution of o_2 and then o_1, are not identical (hence we cannot deduce an immediate contradiction). However, in such case there must be a process p_w that is not an enabled spender for a_1 and, thus,

every possible method invocation o_3 is either read-only or commutes with o_1 and o_2. Suppose the decision step taken by p_w brings the protocol in a 1-valent state (the reasoning for a 0-valent case is analogous). Then the sequential execution of operations o_1, o_2, and then o_3 results in a state q_3 from which p_1 decides 0. In contrast, the sequential execution of o_3 followed by o_1 and then o_2 results in a state q_4 from which p_1 decides 1. By observing that $q_3 = q_4$, we reach a contradiction. □

Putting it all together, we have:

$$k \overset{\text{(Thm. 17)}}{\leq} \mathcal{CN}(T_{S_k}) \quad \text{and} \quad \mathcal{CN}(T_{Q_k}) \overset{\text{(Thm. 18)}}{\leq} k. \tag{5.15}$$

Observing that $S_k \subseteq Q_k \implies \mathcal{CN}(T_{S_k}) \leq \mathcal{CN}(T_{Q_k})$, we can deduce exact synchronization requirements for T_q when q is a synchronization state, i.e., $q \in S_k$:

$$k \leq \mathcal{CN}(T_{S_k}) \leq \mathcal{CN}(T_{Q_k}) \leq k \quad \implies \quad \mathcal{CN}(T_{S_k}) = k. \tag{5.16}$$

Notice that successful completion of specific *approve* operations is *necessary* to reach a synchronization state from which we can wait-free implement consensus for arbitrarily many processes. Concretely, if we start from any state $q \in Q_k$, reaching a state $q' \in Q_{k'}$ with $k' > k$ requires the owner of some account with k enabled spenders to approve other $k' - k - 1$ spenders. This observation suggests that if such *approve* operations—which change the number of enabled spenders for the same account—were not enabled, then the resulting token object would be no stronger than the k-shared asset transfer object.

ERC20 token *vs* k-shared asset transfer. Despite the apparent similarity between ERC20 tokens and k-*AT* objects, our results show that ERC20 tokens are strictly more powerful, in terms of synchronization level, than k-shared asset transfer objects. For ERC20 tokens, any synchronization level can be reached, in principle, by enabling sufficiently many spenders for the same account. Indeed, while the owners of a shared account must be fixed upfront when the contract is deployed, the enabled spenders for an ERC20 account can change dynamically, as the account owner wishes. Similarly, the amount of tokens that each enabled spender is allowed to transfer from that account is flexibly chosen and can be modified at any time by the account owner. In other words, an ERC20 account is more complex than a k-shared account in at

least two dimensions: by supporting an evolving set of spenders and by letting the account owner update the allowances of these spenders. One interpretation of our results is that the dynamic nature of the ERC20 smart contract is reflected in its dynamic consensus number.

5.6 Extension to other token standards

In this section, we discuss how to extend our results to other token standards on Ethereum beyond ERC20. As of the time of writing, several token implementations have been proposed within the Ethereum project, ERC20 being the major reference among all. Some of these proposals are in a testing phase while others have already reached a final phase and have been adopted [72]. In this section we discuss the ERC777 token, the ERC721 non-fungible token, the ERC1155 multi token, and ERC1363 payable token standards.

The ERC777 token standard aims to solve some problems related to ERC20, while maintaining backward compatibility [54]. It defines new features, some of which are similar to those of ERC20, to interact with the tokens. In particular, it defines *operators* to transfer tokens on behalf of another address, similarly to the mechanism enabled by the *allowances* in ERC20, and *hooks*, to simplify the sending process and to offer a single way for sending tokens to any recipient. One of the main differences compared to ERC20 is the mechanism of allowing processes to manage tokens on behalf of others. In ERC20, the *approve* method lets an account owner p define an amount of tokens that the approved process p' can spend on behalf of p. In contrast, an *operator* p' in ERC777 is allowed to spend all the tokens owned by the approving process p. Nevertheless, it is immediate to extend our results to ERC777. Specifically, Algorithms 8 can be adapted by replacing the approved spenders with the corresponding operators.

The ERC721 standard is inspired by ERC20, however, it provides an interface for *non-fungible* tokens [69]. In contrast to standard tokens, all non-fungible tokens are *unique*. In ERC721, every token is uniquely determined by an identifier *tokenId* and can be individually transferred using a *transferFrom* method. Similarly to ERC20, an account owner p can approve other processes to spend tokens on its behalf by invoking the *approve* method, specifying the process p' to be approved and a token identifier *tokenId*. We do not discuss the other methods specified by ERC721, as they fall outside the scope of this work. Although ERC721

defines tokens of different nature compared to ERC20 tokens, we notice that our techniques and results can also be applied, with some adjustment, to this standard. Concretely, Algorithm 8 can be adapted so that it uses a *specific token*, determined by its identifier *tokenId*, which all the participating processes are approved to spend; the winner of this race can then be determined by invoking *ownerOf* with token identifier *tokenId*.

ERC1155 defines a smart-contract interface for managing multiple token types. In particular, it specifies methods that enable the execution of a number of transactions, possibly on different token types, or involving various source and target accounts, within a single method-call. While it is plausible that ERC1155 tokens inherit the synchronization requirements of ERC20 tokens, establishing formal requirements would need an in-depth analysis, based on combinations of accounts, which goes beyond the scope of this work.

Finally, the *Payable Token* standard ERC1363 follows the *approve* and *transferFrom* paradigm of ERC20 tokens, but adds a layer of indirection. Specifically, it allows processes to specify *arbitrary code*, which is executed upon receiving a token through *transfer*, *transferFrom*, or upon completion of an *approve* operation. The possibility of executing an arbitrary contracts precludes establishing exact synchronization *a priori*.

5.7 Discussion

Conclusion. Prior work shows that the asset transfer object, providing the basic functionality of a cryptocurrency, has consensus number 1 [86]. This means that implementing a "plain" cryptocurrency such as Bitcoin [124] does not require synchronization among processes, and hence the consensus layer of Bitcoin could be replaced by a fully asynchronous dissemination protocol, which does not order transactions. This important result however does not apply to blockchains with richer smart-contract support such as Ethereum. In fact, enabling the execution of *arbitrary* smart contracts requires agreement among all blockchain nodes. Nevertheless, it remains open whether *specific* smart contracts need consensus or not, and more generally, which level of synchronization they require.

In this work, we analyze the synchronization requirements of one such smart contract—the ERC20 token standard of Ethereum—through the lens of wait-free implementations, establishing the consensus number of

an associated shared-memory token object. Our results show that an ERC20 token contract may require different levels of synchronization, depending on its state configurations. In other words, the ERC20 token object has a *dynamic* consensus number: when initialized according to the standard [160], its consensus number is 1; however, as soon as an account owner approves other spenders for its account, the consensus number of the object may increase. In fact, there exist executions that modify the state so that the consensus number becomes k, for every k with $1 \leq k \leq n$.

Future directions. Our results imply that while executing arbitrary smart contracts requires consensus among *all* processes, synchronizing a dedicated subset of processes is sufficient for realistic applications such as token contracts. In the case of ERC20 tokens, consensus indeed only needs to be reached among the largest set $\sigma_q(a)$ of enabled spenders for the same account a; importantly, the exact synchronization requirements can be readily deduced from the current object's state q by reading the current balances and allowances. This insight opens up the possibility to deploy realistic smart contracts, such as ERC20 tokens, on more scalable and performant protocols than consensus-based blockchains. Namely, the consistency mechanism could be flexibly adapted, during execution, to require higher or lower coordination among nodes depending on the current state of the smart contract, so that only the minimal synchronization requirements are matched.

We suggest as an interesting open problem to develop distributed protocols meeting the dynamic synchronization requirements of ERC20 tokens. Such protocols could replace the consensus layer of traditional blockchain platforms with a more efficient broadcast method, as shown earlier for asset transfer [50]. This would generally work under asynchrony and yet provide an atomic broadcast functionality among every account owner and its enabled spenders.

Chapter 6

Practical distributed cryptography with general trust

6.1 Introduction

Throughout the last decade, largely due to the advent of blockchains, there has been an ever-increasing interest in distributed systems and practical cryptographic primitives. Naturally, the type of cryptography most suitable for distributed systems is distributed cryptography: independent parties jointly hold a secret key and perform some cryptographic task.

Distributed cryptography finds many applications and deployments today. Threshold signature schemes [64, 24] distribute the signing power among a set of parties. They have been used in state-machine replication (SMR) protocols, a paradigm that increases resilience by replicating applications on multiple hosts, where they serve as unique and constant-size vote certificates [163, 123]. Furthermore, random-beacon and common-coin schemes [56, 35] provide a source of reliable and distributed randomness. In SMR protocols they facilitate, among other tasks, leader election [39, 59, 135] and sharding [164, 102]. As a third example, multiparty computation (MPC) is a cryptographic tool that enables a set of parties to compute a function in a distributed manner,

while keeping their inputs private. It has found applications in protecting digital assets[1], or private keys[2] and cryptocurrency wallets[3], often worth millions of dollars. Applications also include highly sensitive and private data[4], related, for example, to DNA[5] or efforts against human trafficking [58]. Security is, hence, of paramount importance. MPC has been combined with blockchains to enable private computations [22] and fairness [110, 12].

One can thus say that we are in the era of distributed cryptography. However, all currently deployed distributed-cryptographic schemes express their trust assumptions through a number, with a threshold, hence reducing to the setting of *threshold cryptography*, where all parties may misbehave with the same probability. In other words, parties are considered identical, leading to a monoculture-type view of the system. On the other hand, distributed cryptography does not have to be threshold-based. In *general distributed cryptography* the *authorized sets*, the sets of parties sufficient to perform the task, can be arbitrary, and are specified through a general, non-threshold *access structure* (AS). Our position is that general distributed cryptography is essential for distributed systems.

In this chapter we focus on three important distributed-cryptographic primitives for distributed protocols.

Verifiable secret sharing: *Secret sharing* [147] allows a dealer to share a secret in a way that only authorized sets can later reconstruct it. *Verifiable Secret Sharing (VSS)* [81, 132] additionally allows the parties to verify their shares against a malicious dealer.

Common coin: A *common coin* [137, 35] scheme allows a set of parties to calculate a pseudorandom function $\mathcal{U}$, mapping coin names C to uniformly random bits $\mathcal{U}(C) \in \{0, 1\}$ in a distributed way.

Distributed signatures: A *distributed signature* [64, 24] scheme allows a set of parties to collectively sign a message. The parties hold key shares of an unknown private key and create signature shares on individual messages. Once sufficient signature shares are available, they are combined into a unique distributed signature, which can be verified

[1] *Fireblocks*: https://www.fireblocks.com, *Sepior*: https://sepior.com
[2] *Keyless*: https://keyless.io.
[3] *DFNS*: https://www.dfns.co/, *Zengo*: https://zengo.com, *Unbound security*: https://github.com/unboundsecurity
[4] *Sharemind*: https://sharemind.cyber.ee, *Partisia*: https://partisia.com
[5] https://partisia.com/better-data-solutions/surveys

with the standard algorithm of the underlying signature scheme.

Contributions. The goal of this work is to bridge the gap between theory and practice, so as to pave the way for the adoption of general distributed cryptography. Specifically:

- We show how an administrator can specify a general access structure, starting from a collection of sets or a Boolean formula, described in a JSON file. This is then converted into two different encodings, a tree data structure and an MSP, the former used for checking whether a set of parties is authorized and the latter for all algebraic operations. The practicality of both encodings is validated through examples, among which an access structure used in the live Stellar blockchain.
- We recall a general VSS scheme, and then extend the common-coin construction of Cachin, Kursawe, and Shoup [35] into the general-trust model. Moreover, we present a general distributed-signature scheme based on BLS signatures [27], which extends the threshold scheme of Boldyreva [24]. All schemes are in the MSP model, and we provide security definitions and proofs that are appropriate for the general-trust setting.
- We implement and benchmark the aforementioned schemes, both threshold and general versions. We assess the efficiency of the general schemes and provide detailed explanation of the observed behavior, insights, and possible optimizations. The benchmarks include multiple trust assumptions, thereby exploring how they affect the efficiency of the schemes.

Organization. This chapter is organized as follows. Section 6.2 presents the related work and Section 6.3 presents the necessary background. In Section 6.4 we show how general access structures can be efficiently encoded, while in Section 6.5 we discuss and formalize the procedure of *share interpolation* on general access structures. Then, Sections 6.6, 6.7, and 6.8 show the general distributed VSS, common coin, and signature schemes, respectively, together with their security properties and proofs. In Section 6.9 we benchmark the MSP and all three schemes. Finally, in Section 6.10 we mention possible extensions and conclude this chapter.

6.2 Related work

General distributed cryptography. Secret sharing over arbitrary access structures has been extensively studied in theory. The first scheme is presented by Ito, Saito, and Nishizeki [94], where the secret is shared independently for every authorized set. Benaloh and Leichter [21] use monotone Boolean formulas to express the access structure and introduce a recursive secret-sharing construction. Gennaro presents a general VSS scheme [79], where trust is specified as Boolean formulas in disjunctive normal form. As a result, a party receives as many shares as the number of conjunctions it appears in. Choudhury presents general asynchronous VSS and common-coin schemes secure against a computationally-unbounded adversary [48].

Later, the *Monotone Span Program (MSP)* is introduced [97] as a linear-algebraic model of computation. Since then, VSS schemes with general access structures have been formulated in terms of an MSP. In the information-theoretic setting, Cramer, Damgård, and Maurer [51] construct a VSS scheme for any monotone access structure. Nikov *et al.* [128] extend this work to add proactive resharing. A general VSS scheme is also presented by Mashhadi, Dehkordi, and Kiamari [118], which requires multiparty computation for share verification.

A different line of work encodes the access structure using a *vector-space secret-sharing scheme* [29], a special case of an MSP.[6] Specifically, Herranz and Sáez [90] construct a VSS scheme based on Pedersen's VSS [132]. Herranz, Padró, and Sáez [89] construct general distributed RSA signatures based on the threshold RSA scheme of Shoup [149]. Distributed key generation schemes have also been described based on vector-space secret sharing [61, 62].

Attribute-based signatures. ABS schemes [111] are related to distributed signatures. In ABS a signer possesses a number of attributes and can only produce a valid signature if they satisfy a certain predicate on the set of all attributes. ABS schemes are similar to distributed signatures in that they usually encode the attribute predicate as an MSP, but differ from distributed signatures in terms of security requirements (they

[6] A vector-space secret-sharing scheme can be seen as an MSP where each party owns exactly one row. The MSP is, hence, a stronger model as it can encode any access structure [97, 18].

have to consider attribute privacy and adaptive attribute selection), and hence result in more complicated schemes [129, 106].

Common coin schemes. *Common coin* schemes (also called *shared coins*, *coin tossing* schemes, or *random beacons* [65, 56]) model randomness produced in a distributed way. Multiple threshold schemes have been proposed in the literature [56, 35, 137] and are used in practice [65]. Raikwar and Gligoroski [139] present an overview and classification. Our work extends the common-coin scheme of Cachin, Kursawe, and Shoup [35]. The same threshold construction appears in DiSE [3, Figure 6], where it is modeled as a DPRF [125]. The scheme outputs an unbiased value.

6.3 Background

Computational assumptions. Let $G = \langle g \rangle$ be a group of prime order q and $x_0 \xleftarrow{\$} \{0, \ldots, q-1\}$. The *Discrete Logarithm (DL)* assumption is that no efficient probabilistic algorithm, given $g_0 = g^{x_0} \in G$, can compute x_0, except with negligible probability. The *Computational Diffie-Hellman (CDH)* assumption is that no efficient probabilistic algorithm, given $g, \hat{g}, g_0 \in G$, where $\hat{g} \xleftarrow{\$} G$ and $g_0 = g^{x_0}$, can compute $\hat{g}_0 = \hat{g}^{x_0}$, except with negligible probability.

Definition 12 (Gap Diffie-Hellman group [27]). Let $G_1 = \langle g_1 \rangle$ and $G_2 = \langle g_2 \rangle$ be two groups of prime order q, and $h \xleftarrow{\$} G_1$. Let $\alpha, \beta \xleftarrow{\$} \{0, \ldots, q-1\}$.

- The *computational co-Diffie-Hellman (co-CDH)* problem on (G_1, G_2) asks, on input $g_2, g_2^\alpha \in G_2$ and $h \in G_1$, to compute $h^\alpha \in G_1$.
- The *decisional co-Diffie-Hellman (co-DDH)* problem on (G_1, G_2) asks, on input $g_2, g_2^\alpha \in G_2$ and $h, h^\beta \in G_1$, to output TRUE if $\alpha = \beta$ and FALSE otherwise. In the first case we say that $(g_2, g_2^\alpha, h, h^\alpha)$ is a *co-Diffie-Hellman tuple*.
- We say that (G_1, G_2) is a *Gap co-Diffie-Hellman (co-GDH)* group pair if co-DDH is easy but co-CDH is hard to solve on (G_1, G_2). For a more formal definition we refer the reader to [27].

6.4 Specifying and encoding general access structures

An important aspect concerning the implementation and deployment of general distributed cryptography is specifying the Access Structure (AS). We require a solution that is intuitive, so that users or administrators can easily specify it, that facilitates the necessary algebraic operations, such as computing and recombining secret shares, and in the same time offers an efficient way to check whether a given set is authorized.

In this chapter we encode the AS in cryptographic schemes using a tree data structure and a Monotone Span Program (MSP), as we did in Chapter 3. The tree is used for checking whether a given set is authorized, and the MSP for algebraic operations. The AS is first represented as a Monotone Boolean Formula (MBF), which consists of *and*, *or*, and *threshold* operators. A *threshold* operator $\Theta_k^K(q_1, \ldots, q_K)$ specifies that any subset of $\{q_1, \ldots, q_K\}$ with cardinality at least k is authorized, where each q_i can be a party identifier or a nested operator. The *and* and *or* operators are special cases of this. The tree representation is built from the MBF, as explained in Section 3.2.1. The MBF is also used to build the MSP representation, as explained in Section 3.2.2. If the Boolean formula includes in total c operators in the form $\Theta_{d_i}^{m_i}$, then the final matrix M of the MSP that encodes it has $m = \sum_1^c m_i - c + 1$ rows and $d = \sum_1^c d_i - c + 1$ columns, hence size linear in the size of the formula.

We now present some examples of general AS, which we later use in to benchmark the distributed cryptographic schemes.

Example 6. Recent work [70] presents the example of an *unbalanced-AS*[7], where n parties in $\mathcal{P}$ are distributed into two organizations $\mathcal{P}_1$ and $\mathcal{P}_2$, and the adversary is expected to be within one of the organizations, making it easier to corrupt parties from that organization. They specify this with two thresholds, t and k, and allow the adversary to corrupt at most t parties from $\mathcal{P}$ and in the same time at most k parties from $\mathcal{P}_1$ or $\mathcal{P}_2$. For example, we can set $t = \lfloor n/2 \rfloor$ and $k = \lfloor t/2 \rfloor - 1$. Let $n = 9$, $\mathcal{P}_1 = \{p_1, \ldots, p_5\}$, $\mathcal{P}_2 = \{p_6, \ldots, p_9\}$, $t = 4$, and $k = 1$. The access structure (taken as the complement of the adversary structure) is $\mathcal{A}^+ = \{A \subset \mathcal{P} : |A| > 4 \vee (|A \cap \mathcal{P}_1| > 1 \wedge |A \cap \mathcal{P}_2| > 1)\}$. In terms of a monotone Boolean formula, this can be written as $F_{\mathcal{A}} = \Theta_5^9(\mathcal{P}) \vee$

[7] This is a special case of bipartite AS [130].

$\left(\Theta_2^5(\mathcal{P}_1) \wedge \Theta_2^4(\mathcal{P}_2)\right)$. The MSP constructed with the given algorithm has $m = 2n$ rows and $d = t + 2k + 2 = n - 1$ columns.

Example 7. Another classical general AS from the field of distributed systems is the *M-Grid* [114]. Here $n = k^2$ parties are arranged in a $k \times k$ grid and up to $b = k - 1$ Byzantine parties are tolerated. As we saw in Section 3.2.4, an authorized set consists of any t rows and t columns, where $t = \lceil \sqrt{b/2 + 1} \rceil$. Let us set $n = 16$ and, hence, $k = 4$, $b = 3$, and $t = 2$. This means that and any two rows and two columns (twelve parties in total) make an authorized set. The Boolean formula that describes this AS is $F_{\mathcal{A}} = \Theta_2^4\left(\Theta_4^4(R_1), \Theta_4^4(R_2), \Theta_4^4(R_3), \Theta_4^4(R_4)\right) \wedge \Theta_2^4\left(\Theta_4^4(C_1), \Theta_4^4(C_2), \Theta_4^4(C_3), \Theta_4^4(C_4)\right)$, where R_ℓ and C_ℓ denote the sets of parties at row and column ℓ, respectively. We call this access structure the *grid-AS*.

Example 8. The Stellar blockchain supports general trust assumptions for consensus [109]. Each party can specify its own access structure, which is composed of nested threshold operators. We extract[8] the AS of one Stellar validator and show in Figure 6.1 a JSON file that can be used in our general schemes. It can be directly translated into an MSP, enabling general distributed cryptography in or on top of the blockchain of Stellar. The MSP constructed with the presented algorithm has $m = 25$ rows and $d = 15$ columns.

```
{  "select": 6,
   "out-of": [
      {"select": 2, "out-of": ["Blockdaemon1", "Blockdaemon2", "Blockdaemon3"]},
      {"select": 2, "out-of": ["SDF1", "SDF2", "SDF3"]},
      {"select": 2, "out-of": ["WirexSingapore", "WirexUK", "WirexUS"]},
      {"select": 2, "out-of": ["CoinqvestFinland", "CoinqvestHongKong", "CoinqvestGermany"]},
      {"select": 2, "out-of": ["SatoshiPayUS", "SatoshiPaySG", "SatoshiPayDE"]},
      {"select": 2, "out-of": ["FranklinTempleton1", "FranklinTempleton2", "FranklinTempleton3"]},
      {"select": 3, "out-of": ["LOBSTR1", "LOBSTR2", "LOBSTR3", "LOBSTR4", "LOBSTR5"]},
      {"select": 2, "out-of": ["Hercules", "Lyra", "Boötes"]}
]}
```

Figure 6.1. A JSON file that specifies the access structure of the SDF1 validator in the live Stellar blockchain (we use the literals returned by Stellar as party identifiers).

[8] https://www.stellarbeat.io/, https://api.stellarbeat.io/docs/

6.5 Interpolation on general access structures

When working with a threshold access structure, i.e., a t-out-of-n AS, where t is the maximum number of expected corruptions, it is always the case that an authorized set, i.e., a set of $t+1$ parties, uniquely determines the secret *and* the shares of all other parties. Similarly, a maximally unauthorized set, i.e., a set of t parties, and a given secret uniquely determine the shares of all other parties. These facts are used in security proofs: a simulator, for example, given the shares of a maximally unauthorized set and a secret (or, more precisely, exponents of these values) may have to create valid shares (or exponents of shares) for other parties.

With general AS, however, it can be the case that the shares of an authorized set $A \in \mathcal{A}$, or the shares of a maximally unauthorized set $F \in \mathcal{F}$ together with a given secret, do not fully determine all other shares. For example, this can be because a party $p_i \notin F$ can now own more than one secret shares, in a way that adding all the shares of p_i to F makes it authorized, while adding *some* shares of p_i to F keeps in unauthorized. In this section we present an algorithm to compute valid secret shares for parties not in F — where *valid* means that the reconstruction of the secret from any authorized set will result in the same value — given the shares of F and the secret. The same technique can be applied to exponents of these values.

Let $\mathcal{M} = (M, \rho, \boldsymbol{e}_1, \mathcal{P})$ be an MSP over $\mathcal{K}$, with M an $m \times d$ matrix. We have seen in the definition of LSSS that an authorized set A can reconstruct the secret through the equation $\boldsymbol{\lambda}_A \boldsymbol{x}_A = x$. Let $F \in \mathcal{F}$ be a maximally unauthorized set, i.e., there exists no $F' \in \mathcal{F}$ such that $F' \supset F$, and let x be the secret. If the rank of M_F is $d-1$ then all secret shares are uniquely defined. If the rank of M_F is $d-1-k$, for $k \in \mathbb{N}$, then there exist k secret shares (each corresponding to an MSP row), that do not belong to parties in F and are linearly independent from the shares of parties in F. The values of these secret shares can be chosen arbitrarily from $\mathcal{K}$ in the interpolation we wish to perform. These *extra rows* are given to the interpolation algorithm in the form of a set $R \subset \{1, \ldots, m\}$. Formally, the algorithm has the following inputs and outputs.

Inputs:

1. A maximally unauthorized set of parties $F \subset \mathcal{P}$ and their secret

shares $\boldsymbol{x}_F \in \mathcal{K}^{m_F}$, (where m_F is the number of MSP rows owned by parties in F, and might be greater than $|F|$).

2. A set of extra MSP-row indexes $R \subset \{1, \ldots, m\}$, with $\rho(j) \notin F$, for all $j \in R$, and the corresponding secret shares $\boldsymbol{x}_R \in \mathcal{K}^{m_R}$ (where $m_R = |R|$). The sets F and R are such that the rank of the matrix $\binom{M_F}{M_R}$, that consists of the MSP rows either owned by parties in F or corresponding to indexes in R, is $d-1$. Notice that the rows indexed by R can all be chosen to be linearly independent from each other and from the rows owned by parties in F, hence the shares $\boldsymbol{x}_R$ can be chosen uniformly from the underlying field.
3. The secret x that corresponds to the secret shares $\boldsymbol{x}_F$ and $\boldsymbol{x}_R$.
4. An index $j \in [1, \ldots m]$.

Output: Coefficients $\Lambda_j^{(1)} \in \mathcal{K}$ and $\boldsymbol{\Lambda}_j^{(2)} \in \mathcal{K}^{m_F+m_R}$, such that the secret share x_j can be calculated as a linear combination of these coefficients and the input values, that is, $x_j = \Lambda_j^{(1)} x + \boldsymbol{\Lambda}_j^{(2)}(\boldsymbol{x}_F \| \boldsymbol{x}_R)$.

The algorithm works as follows. The given secret shares $\boldsymbol{x}_F \| \boldsymbol{x}_R$ have been computed as

$$\begin{pmatrix} \boldsymbol{x}_F \\ \boldsymbol{x}_R \end{pmatrix} = \begin{pmatrix} M_F \\ M_R \end{pmatrix} \boldsymbol{r}$$

where $\boldsymbol{r} = (x, r_2, \ldots, r_d)$ is unknown, except for the secret x. Since we know x, we can rewrite the previous equation as

$$\begin{pmatrix} x \\ \boldsymbol{x}_F \\ \boldsymbol{x}_R \end{pmatrix} = \begin{pmatrix} \boldsymbol{e}_1 \\ M_F \\ M_R \end{pmatrix} \boldsymbol{r}.$$

We define

$$\overline{M} = \begin{pmatrix} \boldsymbol{e}_1 \\ M_F \\ M_R \end{pmatrix}.$$

Observe that the MSP rows determined by F and R together are still unauthorized, and thus $\boldsymbol{e}_1$ is linearly independent from the rows in $\binom{M_F}{M_R}$. Moreover, by construction of F and R, the rank of $\binom{M_F}{M_R}$ is $d-1$. From these facts we get that $\overline{M}$ has full rank d. Moreover, let $\overline{m}$ be the number of rows in $\overline{M}$.

We now make use of d recombination vectors $\boldsymbol{\lambda}_\ell$, for $\ell \in [1, \ldots d]$. Each recombination vector $\boldsymbol{\lambda}_\ell$ is defined as an $\overline{m}$-vector such that $\boldsymbol{\lambda}_\ell \overline{M} = \boldsymbol{e}_\ell$, where $\boldsymbol{e}_\ell$ is the ℓ-th unit vector (i.e., consists of 0s, except for a 1

in position ℓ) of dimension d. In other words, $\boldsymbol{\lambda}_\ell$ expresses a linear combination of rows of $\overline{M}$ that gives the vector $\boldsymbol{e}_\ell$. Since the rank of $\overline{M}$ is d, all these recombination vectors exist. Additionally, define Λ as the $(d, \overline{m})$ matrix with the d recombination vectors as rows, i.e.,

$$\Lambda = \begin{pmatrix} \boldsymbol{\lambda}_1 \\ \boldsymbol{\lambda}_2 \\ \cdots \\ \boldsymbol{\lambda}_d \end{pmatrix}.$$

Notice that

$$\Lambda \cdot \overline{M} = I_d,$$

where I_d is the (d, d) identity matrix, and, by multiplying both members with $\boldsymbol{r}$,

$$\Lambda \cdot \begin{pmatrix} x \\ \boldsymbol{x}_F \\ \boldsymbol{x}_R \end{pmatrix} = \boldsymbol{r}.$$

By defining as $\boldsymbol{\Lambda}^{(1)}$ the first column of Λ and as $\Lambda^{(2)}$ the last $\overline{m} - 1$ columns, the last equation can be rewritten as

$$\boldsymbol{\Lambda}^{(1)} x + \Lambda^{(2)}(\boldsymbol{x}_F \| \boldsymbol{x}_R) = \boldsymbol{r}.$$

From the last equation we get

$$x_j = \boldsymbol{M}_j \boldsymbol{r} = \boldsymbol{M}_j \boldsymbol{\Lambda}^{(1)} x + \boldsymbol{M}_j \Lambda^{(2)}(\boldsymbol{x}_F \| \boldsymbol{x}_R),$$

or, by setting $\Lambda_j^{(1)} = \boldsymbol{M}_j \boldsymbol{\Lambda}^{(1)}$ and $\boldsymbol{\Lambda}_j^{(2)} = \boldsymbol{M}_j \Lambda^{(2)}$:

$$x_j = \Lambda_j^{(1)} x + \boldsymbol{\Lambda}_j^{(2)}(\boldsymbol{x}_F \| \boldsymbol{x}_R). \tag{6.1}$$

6.6 Verifiable secret sharing

In this section we recall a general Verifiable Secret Sharing (VSS) scheme [90]. It generalizes Pedersen's VSS [81, 132] to the general setting.

Security. The security of a general VSS scheme is formalized by the following properties (in analogy with the threshold setting [81, 132]).

1. Completeness. If the dealer is not disqualified, then all honest parties complete the sharing phase and can then reconstruct the secret.
2. Correctness. For any authorized sets A_1 and A_2 that have accepted their shares and reconstruct secrets z_1 and z_2, respectively, with overwhelming probability it holds that $z_1 = z_2$. Moreover, if the dealer is honest, then $z_1 = z_2 = s$.
3. Privacy. Any unauthorized set F has no information about the secret.

The scheme. The scheme is synchronous and uses the same communication pattern as the standard VSS protocols [81, 132]. Hence complaints are delivered by all honest parties within a known time bound, and we assume a broadcast channel, to which all parties have access. Let $G = \langle g \rangle$ be a group of large prime order q and $h \xleftarrow{\$} G$.

1. *Share(x).* The dealer uses Algorithm 1 to compute the *secret-shares* $\boldsymbol{x} = (x_1, \ldots, x_m) = LSSS.Share(x)$. The dealer also chooses a random value $x' \in \mathbb{Z}_q$ and computes the *random-shares* $\boldsymbol{x'} = (x'_1, \ldots, x'_m) = LSSS.Share(x')$. Let $\boldsymbol{r} = (x, r_2, \ldots, r_d)$ and $\boldsymbol{r'} = (x', r'_2, \ldots, r'_d)$ be the corresponding coefficient vectors. The dealer computes commitments to the coefficients $C_1 = g^x h^{x'} \in G$ and $C_\ell = g^{r_\ell} h^{r'_\ell} \in G$, for $\ell = 2, \ldots d$, and broadcasts them. The *indexed share* (j, x_j, x'_j) is given to party $p_i = \rho(j)$. Index j is included because each p_i may receive more than one such tuples, if it owns more than one row in the MSP. We call a *sharing* the set of all indexed shares $X_i = \{(j, x_j, x'_j) \mid \rho(j) = p_i\}$ received by party p_i.
2. *Verify*(j, x_j, x'_j). For each indexed share $(j, x_j, x'_j) \in X_i$, party p_i verifies that
$$g^{x_j} h^{x'_j} = \prod_{\ell=1}^{d} C_\ell^{M_{j\ell}}, \tag{6.2}$$
where $\boldsymbol{M_j}$ is the j-th row-vector of M and $M_{j\ell}$, for $\ell \in \{1, \ldots d\}$, are its entries.
3. *Complain().* Complaints are handled exactly as in the standard version [81]. Party p_i broadcasts a *complaint* against the dealer for every invalid share. The dealer is disqualified if a complaint is delivered, for which the dealer fails to reveal valid shares.

4. *Reconstruct*(A, X_A). Given the sharings $X_A = \{(j, x_j, x'_j) \mid \rho(j) \in A\}$ of an authorized set A, a combiner party first verifies the correctness of each share. If a share is found to be invalid, reconstruction is aborted. The combiner constructs the vector $\boldsymbol{x}_A = [x_{j_1}, \ldots, x_{j_{m_A}}]$, consisting of the m_A secret-shares of parties in A, and, using Algorithm 1, returns *LSSS.Reconstruct*$(A, \boldsymbol{x}_A)$.

Theorem 19. *Under the discrete logarithm assumption for group G, the above* general VSS *scheme is secure (satisfies completeness, correctness, and privacy).*

Completeness. By inspection of the scheme, honest parties accept their shares. Equation (6.2) will hold because

$$\prod_{\ell=1}^{d} C_\ell^{M_{j\ell}} = g^{\sum_{\ell=1}^{d} r_\ell M_{j\ell}} h^{\sum_{\ell=1}^{d} r'_\ell M_{j\ell}} = g^{\boldsymbol{M_j r}} h^{\boldsymbol{M_j r'}} = g^{x_j} h^{x'_j}$$

Furthermore, by definition of a Q^2 adversary structure, an authorized set A made of honest parties always exists, and, by definition of the MSP, the recombination vector $\boldsymbol{\lambda}_A$ of A always exists. Thus, a party can always reconstruct the secret from the shares of A. □

Correctness. For the first part assume, towards a contradiction, that $z_1 \neq z_2$. Also, let z'_1 and z'_2 be the reconstruction from the random-shares of the two sets. Since the shares are correct, it must hold that $g^{z_1} h^{z'_1} = C_1 = g^{z_2} h^{z'_2}$. Here we show that $g^{z_1} h^{z'_1} = C_1$. For $k = 1, 2$ the secret shares and random shares of parties in the two sets are

$$x_{A_k} = \{x_j^{(k)} \mid \rho(j) \in A_k\} \quad , \quad x'_{A_k} = \{x_j^{\prime(k)} \mid \rho(j) \in A_k\}.$$

Moreover, z_1, z'_1, z_2, z'_2 are calculated by honest parties as

$$z_k = \boldsymbol{\lambda}_{A_k} \boldsymbol{x}_{A_k} \quad , \quad z'_k = \boldsymbol{\lambda}_{A_k} \boldsymbol{x}'_{A_k} \tag{6.3}$$

Written as vectors, where m_k is the number of shares in A_k, for $k = 1, 2$, we have

$$\begin{aligned} \boldsymbol{x}_{A_k} &= (x_{j_1}, \ldots, x_{j_{m_k}}) \\ \boldsymbol{x'}_{A_k} &= (x'_{j_1}, \ldots, x'_{j_{m_k}}) \\ \boldsymbol{\lambda}_{A_k} &= (\lambda_{j_1}, \ldots, \lambda_{j_{m_k}}). \end{aligned} \tag{6.4}$$

We have that

$$
\begin{aligned}
g^{z_1} h^{z_1'} &\overset{(6.3)}{=} g^{\boldsymbol{\lambda}_{A_1} \boldsymbol{x}_{A_1}} h^{\boldsymbol{\lambda}_{A_1} \boldsymbol{x}'_{A_1}} \\
&\overset{(6.4)}{=} g^{\sum_{j:\rho(j)\in A_1} \lambda_j x_j^{(1)}} h^{\sum_{j:\rho(j)\in A_1} \lambda_j x_j'^{(1)}} \\
&= \prod_{j:\rho(j)\in A_1} \left(g^{x_j^{(1)}} h^{x_j'^{(1)}}\right)^{\lambda_j} \\
&\overset{(6.2)}{=} \prod_{j:\rho(j)\in A_1} \left(\prod_{\ell=1}^{d} C_\ell^{M_{j\ell}}\right)^{\lambda_j} \\
&= \prod_{\ell=1}^{d} \prod_{j:\rho(j)\in A_1} C_\ell^{M_{j\ell}\lambda_j} \\
&= \prod_{\ell=1}^{d} C_\ell^{\sum_{j:\rho(j)\in A_1} M_{j\ell}\lambda_j} \\
&= \prod_{\ell=1}^{d} C_\ell^{\boldsymbol{\lambda}_A \boldsymbol{M}_{A\ell}}, \text{ where } \boldsymbol{M}_{A\ell} \text{ is the } \ell\text{-th row of } M_A \\
&\overset{\boldsymbol{\lambda}_A M_A = \boldsymbol{e}_1}{=} \prod_{\ell=1}^{d} C_\ell^{e_{1\ell}}, \text{ where } e_{1\ell} \text{ is the } \ell\text{-th entry of } \boldsymbol{e}_1 \\
&\overset{\boldsymbol{e}_1=[1,0,\ldots,0]}{=} C_1
\end{aligned}
$$

In the same way we get that $g^{z_2} h^{z_2'} = C_1$.

Now, since $z_1 \neq z_2$, it is also the case that $z_1' \neq z_2'$. But from this one can extract the logarithm of h with base g as $\log_g h = (z_1 - z_2)/(z_2' - z_1')$, which is, by assumption, not known.

The second part follows immediately from the fact that the dealer is honest and by simple observation that the output of *Reconstruct()* is $\boldsymbol{\lambda}_A \boldsymbol{x}_A = \boldsymbol{\lambda}_A M_A \boldsymbol{r} = \boldsymbol{e}_1 \boldsymbol{r} = x$, for any authorized set A. □

Privacy. Fix wlog a maximally unauthorized set F consisting of parties controlled by the adversary and let m_F the number of shares owned by parties in F. Assume the dealer has shared a secret x using coefficient vectors $\boldsymbol{r} = (x, r_2, \ldots, r_d)$ and $\boldsymbol{r}' = (x', r_2', \ldots, r_d')$. The view of the adversary consists of the shares $\boldsymbol{x}_F = (x_{j_1}, \ldots, x_{j_{m_F}})$ and $\boldsymbol{x}'_F = (x_{j_1}, \ldots, x_{j_{m_F}})$, where $\rho(j_k) \in F$, for $k \in \{1, \ldots, m_F\}$, and the commitments $C_1 = g^x h^{x'}$ and $C_\ell = g^{r_\ell} h^{r_\ell'}$, for $\ell \in \{2, \ldots, d\}$, created by the

dealer. We then choose arbitrary $\tilde{x} \neq x \in \mathcal{K}$. We want to show that the view of the adversary is consistent with an execution of the VSS where $\tilde{x}$ is the secret shared by the dealer.

Observe that $\tilde{x}$ uniquely defines an $\tilde{x}'$ such that $C_1 = g^{\tilde{x}} h^{\tilde{x}'}$. From Lemma 2, we know there exist coefficient vectors $\tilde{\boldsymbol{r}} = \boldsymbol{r} + (\tilde{x} - x)\boldsymbol{w}$ and $\tilde{\boldsymbol{r}}' = \boldsymbol{r}' + (\tilde{x}' - x')\boldsymbol{w}$, with $\boldsymbol{w} \in \mathcal{K}^d$, that share the secrets $\tilde{x}$ and $\tilde{x}'$, respectively, while the resulting shares $\tilde{\boldsymbol{x}}_F$ and $\tilde{\boldsymbol{x}}'_F$ satisfy $\tilde{\boldsymbol{x}}_F = \boldsymbol{x}_F$ and $\tilde{\boldsymbol{x}}'_F = \boldsymbol{x}'_F$. Notice that the $\boldsymbol{w}$ in the proof of Lemma 2 depends on M_F and not on the coefficient vector, thus it is the same in the equations for $\tilde{\boldsymbol{r}}$ and $\tilde{\boldsymbol{r}}'$.

It remains to show that the commitments $\tilde{C}_\ell = g^{\tilde{r}_\ell} h^{\tilde{r}'_\ell}$, for $\ell \in \{2, \ldots, d\}$, also satisfy $\tilde{C}_\ell = C_\ell$. Let b be the discrete logarithm of h with basis g, i.e., $h = g^b$. Recall that $C_1 = g^x h^{x'} = g^{x+bx'}$ and $C_1 = g^{\tilde{x}} h^{\tilde{x}'} = g^{\tilde{x}+b\tilde{x}'}$. These two equations give

$$x + bx' = \tilde{x} + b\tilde{x}'. \tag{6.5}$$

We now define the vectors $\boldsymbol{c} = \boldsymbol{r} + b\boldsymbol{r}'$ and $\tilde{\boldsymbol{c}} = \tilde{\boldsymbol{r}} + b\tilde{\boldsymbol{r}}'$ and observe that $C_\ell = g^{c_\ell}$ and $\tilde{C}_\ell = g^{\tilde{c}_\ell}$, where c_ℓ and $\tilde{c}_\ell$ are the entries of $\boldsymbol{c}$ and $\tilde{\boldsymbol{c}}$, respectively. It is thus enough to show that $\boldsymbol{c} = \tilde{\boldsymbol{c}}$. We have that

$$\begin{aligned} \boldsymbol{c} = \tilde{\boldsymbol{c}} &\Leftrightarrow \boldsymbol{r} + b\boldsymbol{r}' = \boldsymbol{r} + (\tilde{x} - x)\boldsymbol{w} + b\boldsymbol{r}' + b(\tilde{x}' - x')\boldsymbol{w} \\ &\Leftrightarrow (\tilde{x} - x)\boldsymbol{w} + b(\tilde{x}' - x')\boldsymbol{w} = \boldsymbol{0} \\ &\overset{\boldsymbol{w} \neq \boldsymbol{0}}{\Leftrightarrow} \tilde{x} - x + b(\tilde{x}' - x') = 0, \end{aligned}$$

which holds from (6.5). □

6.7 Common coin

The scheme extends the threshold coin scheme of Cachin, Kursawe, and Shoup [35] to accept any general access structure. It works on a group $G = \langle g \rangle$ of prime order q and uses the following cryptographic hash functions: $H : \{0,1\}^* \to G$, $H' : G^6 \to \mathbb{Z}_q$, and $H'' : G \to \{0,1\}$. The first two, H and H', are modeled as random oracles. The idea is that a secret value $x \in \mathbb{Z}_q$ uniquely defines the value $\mathcal{U}(C)$ of a publicly-known coin name C as follows: hash C to get an element $\tilde{g} = H(C) \in G$, let $\tilde{g}_0 = \tilde{g}^x \in G$, and define $\mathcal{U}(C) = H''(\tilde{g}_0)$. The value x is secret-shared among $\mathcal{P}$ and unknown to any party. Parties can create coin shares using their secret shares. Any party that receives enough coin shares can then obtain $\tilde{g}_0$ by interpolating x in the exponent.

Security. The security of a general common-coin scheme is captured by the following properties (analogous to threshold common coins [35]).

1. Robustness. Except with negligible probability, the adversary cannot produce a coin C and valid coin shares for an authorized set, such that and their combination outputs a value different than $\mathcal{U}(C)$.
2. Unpredictability. Unpredictability is defined through the following game. The adversary corrupts, w.l.o.g, a maximally unauthorized set F. It interacts with honest parties according to the scheme and in the end outputs a coin name C, which was not submitted for coin-share generation to *any* honest party, as well as a coin-value prediction $b \in \{0, 1\}$. Then, the probability that $\mathcal{U}(C) = b$ should not be significantly different from $1/2$.

The scheme. It consists of the following algorithms.

1. *KeyGen()*. A dealer chooses uniformly an $x \in \mathbb{Z}_q$ and shares it among $\mathcal{P}$ using the MSP-based LSSS from Algorithm 1, i.e., $\boldsymbol{x} = (x_1, \ldots, x_m) = LSSS.Share(x)$. The secret key x is destroyed after it is shared. We call a *sharing* the set of all key shares $X_i = \{(j, x_j) \mid \rho(j) = p_i\}$ received by party p_i. The verification keys $g_0 = g^x$ and $g_j = g^{x_j}$, for $1 \leq j \leq m$, are made public.
2. *CoinShareGenerate(C)*. For coin C, party p_i calculates $\tilde{g} = H(C)$ and generates a coin share $\tilde{g}_j = \tilde{g}^{x_j}$ for each key share $(j, x_j) \in X_i$. Party p_i also generates a *proof of correctness* for each coin share, i.e., a proof that $\log_{\tilde{g}} \tilde{g}_j = \log_g g_j$. This is the Chaum-Perdersen proof of equality of discrete logarithms [44] collapsed into a non-interactive proof using the Fiat-Shamir heuristic [75]. For every coin share $\tilde{g}_j$ a valid proof is a pair $(c_j, z_j) \in \mathbb{Z}_q \times \mathbb{Z}_q$, such that

$$c_j = H'(g, g_j, h_j, \tilde{g}, \tilde{g}_j, \tilde{h}_j), \text{ for } h_j = g^{z_j}/g_j^{c_j} \ , \ \tilde{h}_j = \tilde{g}^{z_j}/\tilde{g}_j^{c_j}. \quad (6.6)$$

   Party p_i computes such a proof for coin share $\tilde{g}_j$ by choosing s_j at random, computing $h_j = g^{s_j}, \tilde{h}_j = \tilde{g}^{s_j}$, obtaining c_j as in (6.6), and setting $z_j = s_j + x_j c_j$.
3. *CoinShareVerify*$(C, \tilde{g}_j, (c_j, z_j))$. Verify the proof above.
4. *CoinShareCombine()*. Each party sends its coin sharing $\{(j, \tilde{g}_j, c_j, z_j) \mid \rho(j) = p_i\}$ to a designated combiner. Once valid coin shares from an authorized set A have been received, find the

recombination vector $\boldsymbol{\lambda}_A$ for set A and calculate $\tilde{g}_0 = \tilde{g}^x$ as

$$\tilde{g}_0 = \prod_{j|\rho(j)\in A} \tilde{g}_j^{\boldsymbol{\lambda}_A[j]}, \tag{6.7}$$

where the set $\{j \mid \rho(j) \in A\}$ denotes the MSP indexes owned by parties in A. The combiner outputs $H''(\tilde{g}_0)$.

Theorem 20. *In the random oracle model, the above* general common coin *scheme is secure (robust and unpredictable) under the assumption that CDH is hard in G.*

Proof. The proof for the general coin construction follows the lines of the threshold coin scheme [35]. In a high level, we assume an adversary that can predict the value of a coin with non-negligible probability and show how to use this adversary to solve the CDH problem in G. The simulator, which is given g, the public key $g_0 = g^x$, and some $\hat{g}$ as a CDH instance, programs the random oracle H to output $\hat{g}$ for some hash query $\hat{C}$ of the adversary. If the adversary succeeds in predicting the value of $\hat{C}$, then the simulator can extract $\hat{g}_0 = \hat{g}^x$, the solution to its CDH input, from the hash query $H''(\hat{g}_0)$ made by the adversary.

We additionally have to handle specific issues that arise from the interpolation with general access structures. Specifically, the simulator, given the shares of F, has to create valid shares (hidden in the exponent) for other parties. As opposed to the threshold case, it can be the case that the shares of F, together with the secret x, do not fully determine all other shares. The details have been described in Section 6.5.

Robustness follows from the soundness of the interactive proof of equality of the discrete logarithms. Moreover, the underlying access structure is Q^2, hence there will be enough honest parties to combine the shares and interpolate the coin value.

The rest of this proof concerns *unpredictability*. We assume an adversary that can predict the value of a coin with non-negligible probability and show how to use this adversary to solve CDH. To successfully attack CDH, it is enough to construct an algorithm that, on input elements $g, \hat{g}, g_0 \in G$, where $\hat{g} \xleftarrow{\$} G$ and $g_0 = g^{x_0}$, outputs a list that contains $\hat{g}_0 = \hat{g}^{x_0}$ with non-negligible probability [148]. The adversary makes a series of queries for coins $C_1, \ldots, C_t$ for a polynomially large t, and tries to predict the value of the target coin $\hat{C}$. We assume that $\hat{C} = C_s$, for a random $s \in \{1, \ldots, t\}$, which decreases our advantage by a factor of t. For the target coin, let $\hat{g} = H(\hat{C})$ and $\hat{g}_j = \hat{g}^{x_j}$

Only for this part of the proof, we let the adversary corrupt a set $T \supseteq F$, as long as $T \notin \mathcal{A}$, i.e., T is a maximal superset of F that remains unauthorized. This is w.l.o.g: if the adversary cannot predict the coin from T, it cannot predict it from F either. The algorithm simulates the view for the adversary as follows. For party p_i in T we choose its key shares x_j, where $\rho(j) = p_i$, uniformly from $\mathbb{Z}_q$. The verification keys can then be computed as $g_j = g^{x_j}$. For the rest of the verification keys the idea is to use the verification keys we just calculated and g_0, and perform an interpolation in the exponent. However, as explained in Section 6.5, for these to be uniquely determined, the shares of T (called F in Section 6.5) and of some extra indexes R are required. The set of row indexes R is chosen arbitrarily, under the conditions described in Section 6.5. The shares x_j, where $j \in R$, are also chosen uniformly at random, and the corresponding verification keys are again $v_j = g^{x_j}$, where $j \in R$.

We can now use (6.1) with shares $\boldsymbol{x}_T$ and $\boldsymbol{x}_R$ and the secret x raised to g_2:

$$v_j = v^{\Lambda_j^{(1)}} \cdot \prod_{\substack{\ell \text{ such that} \\ \rho(\ell) \in T \vee \ell \in R}} v_\ell^{\Lambda_{j\ell}^{(2)}}. \tag{6.8}$$

After the verification keys are chosen, we simulate the interaction with the adversary as follows. In the random oracle model, the adversary queries H to obtain $\tilde{g}$ or $\hat{g}$ and the simulator can respond to these queries as it wishes. For coins $C \neq \hat{C}$, the simulator chooses $r \in \mathbb{Z}_q$ at random and sets $\tilde{g} = g^r$ as the value of H at point C. The coin shares for all honest parties can be calculated as $\tilde{g}_j = g_j^r$, where $\rho(j) \notin T$.

The proof of correctness for each coin share can be simulated by invoking the random oracle model for H'. When an honest party is supposed to create a coin share $\tilde{g}_j$, the simulator chooses $c_j, z_j \in \mathbb{Z}_q$ at random, and sets the output of H' at point $(g, g_j, g^{z_j} g_j^{-c_j}, \tilde{g}, \tilde{g}_j, \tilde{g}^{z_j} \tilde{g}_j^{-c_j})$ to be c. Except with negligible probability, the simulator has not already defined the output of H' at this point, so this part of the simulation succeeds.

For the target coin $\hat{C}$ we set $H(\hat{C}) = \hat{g}$. By construction of T, the adversary is not allowed to ask honest parties for coin shares, thus the simulator never has to produce any valid shares. Observe that the adversary, in order to make the prediction $b \in \{0, 1\}$ for $\hat{C}$, must query H'' at point $\hat{g}_0$. Hence, when it terminates we output the list of all these queries — by assumption it will contain the solution to CDH with a non-

negligible probability. The simulation is perfect, since all the shares and verification keys have the same distribution as in an actual execution of the protocol, except for a negligible probability that our zero-knowledge simulations fail. □

6.8 Distributed signatures

In a distributed signature scheme parties hold *key shares* of an unknown private key, created with a *KeyGen()* algorithm, run either by a trusted party or in a distributed manner. Using these, they create *signature shares* on individual messages, using algorithm *Sign()*. Once sufficient signature shares are available, they can be combined into a unique *distributed signature*, using algorithm *SigShareCombine()*. Both signature shares and the distributed signature can be verified as a standard signature of the underlying signature scheme, using *SigShareVerify()* and *Verify()*, respectively.

We now show a general distributed-signature scheme based on the BLS signature scheme [27], which extends the threshold scheme of Boldyreva [24] in the general-trust setting. It works with a co-GDH group pair $G_1, G_2 = \langle g_2 \rangle$ with $|G_1| = |G_2| = q$, for q prime.

Security. In accordance with threshold distributed signatures [149], we demand two basic requirements from general distributed signatures, robustness and unforgeability.

1. Robustness. We say that the scheme is *robust* if the adversary cannot prevent the successful termination (creation of a valid general distributed signature).
2. Unforgeability. It is defined through the following game. The adversary corrupts an adversary set $F \in \mathcal{F}$ of its choice. In the dealing phase the adversary receives all the private-key shares owned by parties in F, as well as the public key and all verification keys. After the dealing phase the adversary submits signing requests for messages of its choice to the honest parties. We say that the adversary *forges* a signature if at the end of the game it outputs a valid signature on a message that was not submitted as a signing request to *any* honest party (together with F this would have given the adversary enough signature shares to reconstruct the distributed signature). The scheme is *unforgeable* if it is infeasible for

the adversary to forge a signature.

The scheme. It consists of the following algorithms.

1. *KeyGen()*. A trusted dealer chooses random $x \in \mathbb{Z}_q$ as the global and unknown to all parties private key and shares it among $\mathcal{P}$ using the MSP-based LSSS from Algorithm 1, i.e., $\boldsymbol{x} = (x_1, \ldots, x_m) = LSSS.Share(x)$. The public key is $v = g_2^x \in G_2$ and the verification keys are $v_j = g_2^{x_j} \in G_2$, for $1 \leq j \leq m$, and they are published. The *sharing* $X_i = \{(j, x_j) \mid \rho(j) = p_i\}$ is given to p_i.
2. *Sign*(μ, X_j). For each indexed share $(j, x_j) \in X_i$, the owner party p_i calculates an indexed share of the signature (j, σ_j), where $\sigma_j = H(\mu)^{x_j} \in G_1$.
3. *SigShareVerify*(μ, σ_j, v, v_j). Verify that $(g_2, v_j, H(\mu), \sigma_j)$ is a co-Diffie-Hellman tuple.
4. *SigShareCombine*$((j_1, \sigma_{j_1}), \ldots, (j_{m_A}, \sigma_{j_{m_A}}))$. Once the indexed signature shares $\sigma_{j_1}, \ldots, \sigma_{j_{m_A}}$ from an authorized group A have been received, recover the distributed signature as $\sigma = \prod_{j \in A} \sigma_j^{\boldsymbol{\lambda}_A[j]}$, where $\boldsymbol{\lambda}_A[j]$ are the entries of the recombination vector that corresponds to A.
5. *Verify*(μ, σ, v). Verify that $(g_2, v, H(\mu), \sigma)$ is a co-Diffie-Hellman tuple.

Theorem 21. *Assuming that standard BLS signatures are secure, the general distributed signature scheme above is secure (robust and unforgeable).*

Robustness. Because $\mathcal{A}$ is Q^2, there exists an authorized set A that consists entirely of honest parties. Moreover, only valid signatures, made with a party's private key share, can pass the verification of algorithm *SigShareVerify()*. Thus, a combiner can verify and use the signature shares of A in algorithm *SigShareCombine()* to create a valid distributed BLS signature. □

Unforgeability. We show that the general distributed signature scheme is simulatable. Simulatability, together with the unforgeability of the standard BLS scheme, imply unforgeability for the general distributed signature scheme [80, Definition 3]. Simulatability means that a simulator, on input the public key v, a message μ with signature σ, and the key shares x_j of parties in F, i.e., $\rho(j) \in F$, can simulate the view for

the adversary that is polynomially indistinguishable from an execution of the real protocol that outputs σ as the signature of μ, and where the adversary has key shares x_j, where $\rho(j) \in F$. Intuitively, this shows that an adversary who sees all the private information of parties in F and the signature on a message μ could generate by itself all the public information of the protocol.

The simulator works as follows. First, it has to provide valid verification keys for all parties and all their shares. For parties in F, the simulator can use the given shares x_j, where $\rho(j) \in F$, to compute the verification keys. The rest of the shares are interpolated from the shares x_j. However, as explained in Section 6.5, for these to be uniquely determined, some extra indexes R are required. The set of row indexes R is chosen arbitrarily, under the conditions described in Section 6.5, and the shares that correspond to the indexes in R are chosen uniformly at random. For sets F and R, the simulator computes the verification keys as $v_j = g_2^{x_j}$, where $\rho(j) \in F$ or $j \in R$. For any other p_j the simulator uses the interpolation algorithm described in Section 6.5, with input sets F and R, and with shares $\boldsymbol{x}_F$ and $\boldsymbol{x}_R$ and the secret x raised to g_2, calculating v_j exactly as we did in (6.8).

Second, the simulator also has to respond to the adversary's signature queries. Following exactly the same techniques, the simulator can generate all the signature shares given the standard BLS signature σ of message m.

Finally, for any row $j \in \{1, \ldots, m\}$ of the MSP, the verification key $v_j = g_2^{x_j}$ and the signature share $\sigma_j = H(\mu)^{x'_j}$ will satisfy $x_j = x'_j$. For j such that $\rho(j) \in F$ or $j \in R$ this holds because the simulator used a known x_j to calculate these values, while for any other j this holds from the MSP interpolation. Hence, $(g_2, v_i, H(m), \sigma_i)$ is a valid co-Diffie-Hellman tuple and the signature shares will be verified. Moreover, the interpolated key shares have the same distribution as if produced by the real dealer. The view of the adversary is thus statistically indistinguishable from an execution of the real protocol. □

6.9 Evaluation

In this section we compare the polynomial-based and MSP-based encodings for trust assumptions, and benchmark the presented schemes on multiple general trust assumptions. To this goal, we benchmark each scheme on four configurations, resulting from different combinations of

encoding and access structure (AS), as seen in Table 6.1. Notice that the first two describe the same threshold AS, encoded once by a polynomial and once an MSP. With the first two configurations we investigate the practical difference between polynomial-based and MSP-based encoding of the same access structure. The last three configurations measure the efficiency we sacrifice for more powerful and expressive AS.

Table 6.1. Evaluated configurations and corresponding MSP dimensions. Configurations with general AS encode it as an MSP (necessary for algebraic operations, such as sharing and reconstruction) and as a tree (for checking whether a set of parties is authorized). The general AS have been presented in Examples 6 and 7.

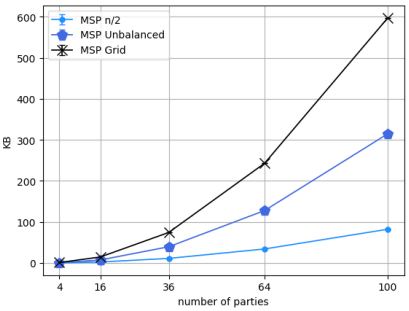
			MSP dimensions	
Configuration	Encoding	AS	m	d
polynomial $(n+1)/2$	polynomial	$\lceil \frac{n+1}{2} \rceil$-of-$n$	-	-
MSP $(n+1)/2$	MSP+tree	$\lceil \frac{n+1}{2} \rceil$-of-$n$	n	$\lceil \frac{n+1}{2} \rceil$
MSP Unbalanced	MSP+tree	*unbalanced*	$2n$	$n-1$
MSP Grid	MSP+tree	*grid*	$2n$	$2(n+t-k) \approx 2n$

We implement all presented schemes in C++. The benchmarks only consider CPU complexity, by measuring the time it takes a party to execute each algorithm. Network latency, parallel share verification, and communication-level optimizations are not considered, as they are independent to the encoding of the AS. All benchmarks are made on a virtual machine running Ubuntu 22.04, with 16 GB memory and 8 dedicated CPUs of an AMD EPYC-Rome Processor at 2.3GHz and 4500 bogomips. The number of parties n is always a square, for *grid-AS* to be well-defined, and we report mean values and standard deviation over 100 runs with different inputs.

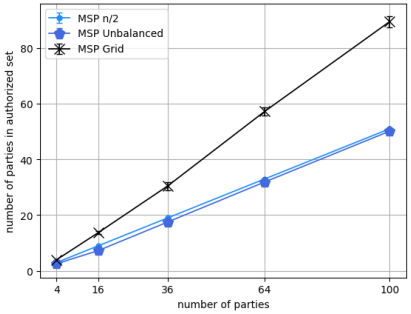
6.9.1 Benchmarking basic properties of the MSP

We first measure the space (size in KB) needed to store the MSP that describes each general AS. The MSP needs to be stored by every party, as it used to compute the recombination vector. We remark that, by construction of Algorithm 2, an AS described with a large number of nested operators results in a spare MSP matrix. The result for different values of n is shown in Figure 6.2a.

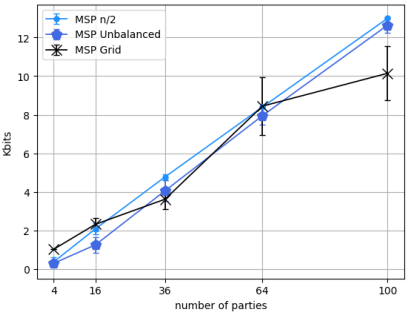
We next measure the size (as number of parties) of authorized sets for each AS. Authorized sets are obtained in the following way. Starting



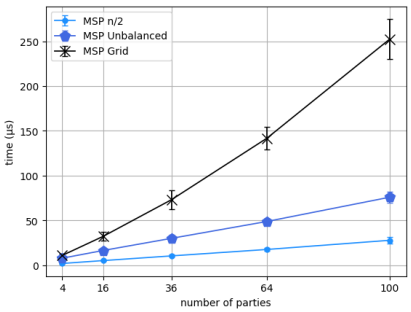
(a) Size (in KB) of the MSP



(b) Size (number of parties) of authorized set



(c) Bit length of recombination vector



(d) Time to check for authorized set

Figure 6.2. Benchmarking basic properties of the MSP, for a varying number of parties. In 6.2d, the tree representation of the AS is used and the set is chosen uniformly among all subsets of $\mathcal{P}$.

from an empty set, add a party chosen uniformly from the set of all parties, until the set becomes authorized. This simulates an execution where shares arrive in an arbitrary order, and may result in authorized sets that are not minimal, in the sense that they are supersets of smaller authorized sets, but contain redundant parties. We repeat this experiment 1000 times and report the average size in Figure 6.2b. For the $\lceil \frac{n+1}{2} \rceil$-of-$n$ AS, of course, authorized sets are always of size $\lceil \frac{n+1}{2} \rceil$. For the *unbalanced-AS* they slightly smaller, and for the *grid-AS* they are significantly larger, as they contain full rows and columns of the grid.

We next measure the bit length of the recombination vector. This

is relevant because the schemes involve interpolation in the exponent, exponentiation is an expensive operation, and a shorter recombination vector results in fewer exponentiations. We observe in Figure 6.2c that the complexity of the AS (in terms of the size of the Boolean formula or the JSON file that describes it) does not necessarily affect the bit length of the recombination vector. There are two important observations to explain Figure 6.2c. First, each entry of the recombination vector that corresponds to a redundant party is 0, as that share does not contribute to reconstruction. Second, we observe through our benchmarks that, when the MSP is sparse and has entries with short bit length, then the recombination vector also has a short bit length.

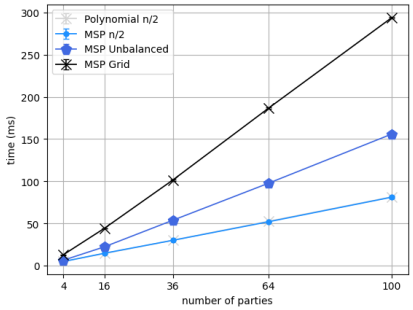
Finally, in Figure 6.2d we report the time it takes to check whether a given set is authorized. This set is chosen uniformly at random among all subsets of $\mathcal{P}$ and an average is taken over 1000 sets. As explained in Section 6.4, the algorithm that checks for authorized sets uses the tree representation of the AS, as it is more efficient than using the MSP.

6.9.2 Running time of verifiable secret sharing

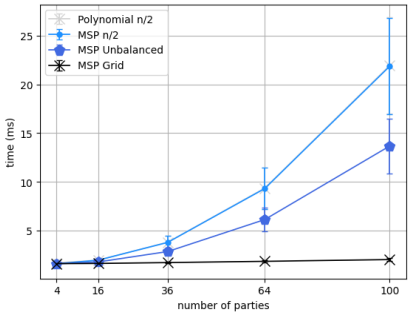
We implement and compare the MSP-based scheme of Section 6.6 with Pedersen's VSS[9] [132], which we refer to as *general VSS* and *threshold VSS*, respectively. For the *Share()* algorithm we report the time it takes a dealer to share a random secret $s \in \mathbb{Z}_q$, for *Verify()* the average time it takes a party to verify *one* of its shares (notice that in the general scheme a party may receive more than one shares), and for *Reconstruct()* the time it takes a party to reconstruct the secret from an authorized group. For the latter, the group is assumed authorized, i.e., we do not include the time to check whether it is authorized, as this is efficiently done using the tree encoding. The results are shown in Figure 6.3.

The first conclusion (comparing the first two configurations in Figures 6.3a and 6.3b) is that the MSP-based and polynomial-based operations are equally efficient, when instantiated with the same AS. The only exception is the *Reconstruct()* algorithm, shown in Figure 6.3c, where general VSS is up to two times slower. This is because computing the recombination vector employs Gaussian elimination, which has cubic time complexity. Nevertheless, the reconstruction of the secret only involves operations in field $\mathcal{K}$, which is relatively fast — *Reconstruct()* is an order of magnitude faster than *Verify()*.

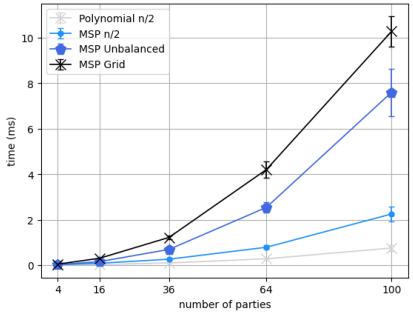
[9] Polynomial evaluation is done without the DFT optimization.



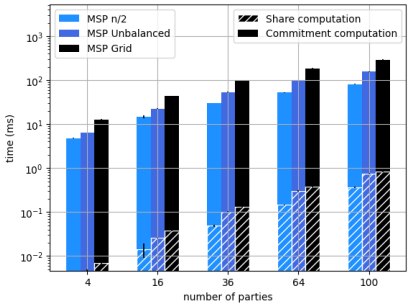
(a) Time taken for *Share()*



(b) Time taken for *Verify()*



(c) Time taken for *Reconstruct()*



(d) Time taken by *Share()* calculating shares and commitments

Figure 6.3. Time taken by each algorithm in the threshold and general VSS for a varying number of parties. Figure 6.3a measures the time for a dealer to share a secret, 6.3b the time for a party to verify *one* of its shares, and 6.3c the time for a party to reconstruct the secret. Figure 6.3d compares the time (in logarithmic scale) needed by *Share()* to compute the shares against the time to compute commitments to the shares.

The second conclusion (comparing the last three configurations, i.e., the ones that use general trust) is that general VSS is moderately affected by the complexity of the AS. For *Share()*, shown in Figure 6.3a, more complex AS incur a slowdown because a larger number of shares and commitments have to be created. *Reconstruct()*, in Figure 6.3c, is also slower with more complex AS, because it performs Gaussian elimination on a larger matrix. We conclude this is the only part of general VSS that cannot be made as efficient as in threshold VSS. On the other hand, *Verify()*, in Figure 6.3b, exhibits an interesting behavior: the more complex the AS, the faster it is on average to verify *one* share. This might seem counter-intuitive, but can be explained from the observations of Section 6.9.1; more complex AS result in an MSP with many 0-entries, hence the exponentiations of (6.2) are faster.

An observation that might be useful for future optimizations is that almost the entire time of *Share()* is spent computing commitments; the dealer computes d commitments, which require $2d$ exponentiations. As shown in Figure 6.3d, the computation of shares is orders of magnitude faster. Another possible optimization is to parallelize algorithm *Share()*, since the computation of shares and commitments is independent of each other.

6.9.3 Running time of common coin

We implement the general scheme of Section 6.7 and the threshold coin scheme from [35]. For both schemes G is instantiated as an order-q subgroup of $\mathbb{Z}_p$, where $p = qm+1$, for q a 256-bit prime, p a 3072-bit prime, and $m \in \mathbb{N}$. These lengths offer 128-bit security and are chosen according to current recommendations for discrete logarithm prime fields [68, Chapter 4.5.2] [10]. The arithmetic is done with NTL [151]. The hash functions H, H', H'' use the openSSL implementation of SHA-512 (so that it's not required to expand the digest before reducing modulo the 256-bit q [150, Section 9.2]).

The results are shown in Figure 6.4. We only show the benchmark of *CoinShareCombine()*, because *KeyGen()* behaves very similar to *Share()* in the VSS, and *CoinShareGenerate()* and *CoinShareVerify()* are identical in the general and threshold scheme (the average time to create and verify, respectively, one coin share was always approximately 4.5ms). In Section 6.9.2 we observed that *Reconstruct()* was slower for the general

[10] Summary of recommendations from multiple organizations: https://www.keylength.com/en/3

scheme, because it involved no exponentiations and the cost of matrix manipulations dominated the running time. Here, however, *CoinShare-Combine()* runs similarly in all cases, as the exponentiations in (6.7) become dominant. As a matter of fact, the general scheme is sometimes faster. This is because complex AS often result in recombination vectors with shorter bit length, as shown in Section 6.9.1, hence exponentiations are faster.

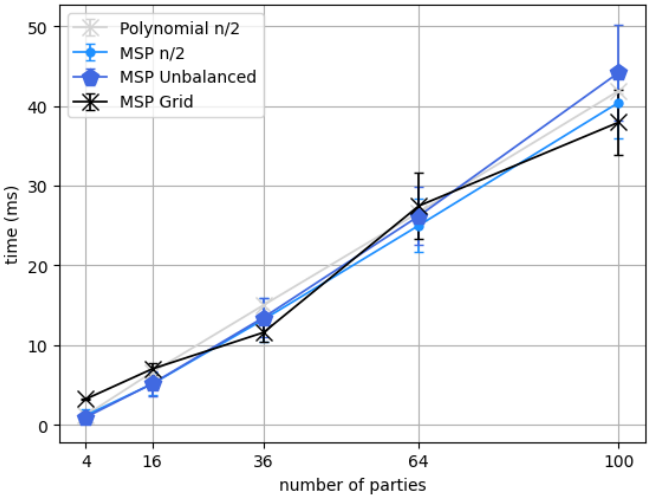


Figure 6.4. Time taken by *CoinShareCombine()* in the threshold and general coin for a varying number of parties.

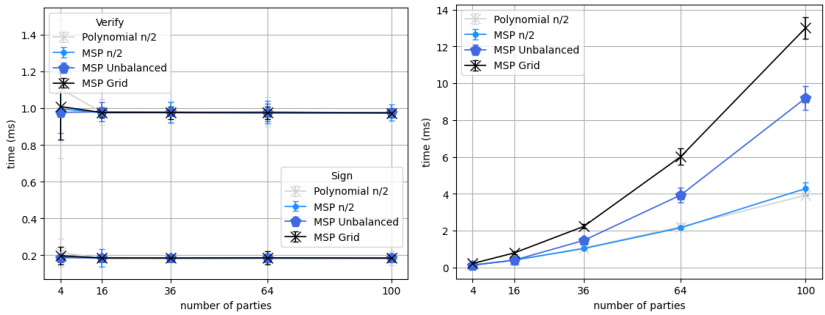
6.9.4 Running time of distributed signatures

We have implemented the general distributed signature scheme from Section 6.8. Our extension for generalized operations are made on the *bls* library[11], which in turn uses *mcl*[12] for pairing operations. The security of these libraries has been reviewed [136] on behalf of the Ethereum Foundation. The benchmarks are done over BLS12-381[17], a widely used pairing-friendly curve offering 128 bits of security [53, Section 4.1].

[11] https://github.com/herumi/bls, commit 64d13b9
[12] https://github.com/herumi/mcl, version 1.40

The observations are similar to those for the previous schemes. Creating and verifying a single signature share, as shown in Figure 6.5a, does not depend on the scheme or the complexity of the AS, hence the corresponding algorithms run in constant time. On the other hand, *SigShareCombine()*, as shown in Figure 6.5b, is moderately affected by the complexity of the AS: similar to *Reconstruct()* in the VSS and different from *CoinShareGenerate()* in the common-coin scheme, *SigShareCombine()* does not involve exponentiations, but only calculation of the recombination vector and multiplication of elliptic curve points by constants. For this reason the computation of the recombination vector dominates running time, and *SigShareCombine()* becomes slower on more complex AS.

We finally remark that the general distributed signature scheme is considerably more efficient than the state-of-the-art solution: assuming we have m signatures from an authorized set, the state-of-the-art would require each party to verify all of them. When a scheme with general trust is available, the signatures can first be combined. The cost of combining them remains in all cases much lower than the cost of verifying each one individually.



(a) Time taken for *Sign()* and *Verify()* **(b)** Time taken for *SigShareCombine()*

Figure 6.5. Time taken by each algorithm in the threshold and general distributed signature scheme for a varying number of parties. Figure 6.5a measures the time for a party to create and verify *one* signature share and 6.5b the time to combine an authorized set of signature shares.

6.10 Discussion

Conclusion. In this chapter we provide the first implementation and practical assessment of distributed cryptography with general trust. We fill all gaps on implementation details and show how a system can be engineered to support general distributed cryptography. We describe, implement, and benchmark distributed cryptographic schemes, specifically, a verifiable secret-sharing scheme, a common-coin scheme, and a distributed signature scheme (as a generalization of threshold signatures), all supporting general trust assumptions. For completeness, we also present the security proofs for all general schemes and handle specific cases that arise from the general trust assumptions (see Theorem 20). Our results suggest that practical access structures can be used with no significant efficiency loss. It can even be the case (VSS share verification, Figure 6.3b) that operations are on average faster with complex trust structures encoded as Monotone Span Programs (MSP). We nevertheless expect future optimizations, orthogonal to our work, to make MSP operations even faster. Similar optimizations have already been discovered for polynomial evaluation and interpolation [157]. We expect that our work will improve the understanding and facilitate the wider adoption of general distributed cryptography.

Future work. Distributed key generation (DKG) is a significant component in distributed cryptographic schemes. It eliminates the strong assumption of a trusted dealer by distributing this task among the parties. The basic idea is that each party runs an instance of VSS in parallel, sharing a random secret, and then locally adds the shares of the instances that successfully terminated (i.e., their dealer did not get disqualified). The shared secret, which never becomes known to any party, is uniquely determined as the sum of the random secrets of the instances that terminated. This technique can be used in MSP-based DKG protocols, as well, although we leave the formal description of an MSP-based DKG scheme as future work. This boils down to the linearity of MSPs: adding two share vectors $\boldsymbol{z}_1 = M\boldsymbol{r}_1$ and $\boldsymbol{z}_2 = M\boldsymbol{r}_2$, where $\boldsymbol{r}_1[1] = x_1$ and $\boldsymbol{r}_2[1] = x_2$, and then interpolating from some authorized set A will always result in the sum of the two shared secrets, i.e., $\boldsymbol{\lambda}_A(\boldsymbol{z}_1 + \boldsymbol{z}_2) = \boldsymbol{\lambda}_A M(\boldsymbol{r}_1 + \boldsymbol{r}_2) = x_1 + x_2$.

Chapter 7

Practial large-scale distributed randomness generation

This chapter presents a simple and efficient *common-coin* protocol, which can be used for generating the randomness required in asynchronous total-order broadcast (ATOB) protocols. The corresponding full paper [7] additionally presents a *seed-generation* protocol, which can be used in proof-of-stake total-order broadcast protocols when unpredictable nonce values are needed [59, 82]. Both protocols are secure in a proof-of-stake setting with dynamically changing stake. They can be plugged into existing ATOB protocols and will turn them into practical and efficient ATOB protocols with dynamic stake and with proactive key refreshing. The full paper [7] also shows how to use our common-coin protocol in *DAG Rider* [99], a state-of-the-art DAG-based ATOB protocol.

7.1 Introduction

State of the art. It is well known that asynchronous total-order broadcast (ATOB) cannot be deterministic [76]. The necessary randomness is usually modelled as a *common coin* scheme [137], informally defined as a source random values observable by all participants but

unpredictable for the adversary [35]. Common coins are most practically implemented using threshold cryptography [39, 65, 135, 35]. This approach has many benefits. It is conceptually simple and efficient, it achieves optimal resilience $t < n/3$, where n parties run the protocol, and it results in a *perfect coin*, meaning that it is uniformly distributed and agreed-upon with probability 1. The drawback, however, is that it requires a trusted setup or an asynchronous distributed key generation (ADKG) protocol. Current state of the art ADKG protocols [57, 1, 2] have communication cost of $O(n^3)$.

Given that state-of-the-art ATOB protocols have communication complexity $O(n^2)$, or even amortized $O(n)$, it is evident that the communication cost of ADKG becomes the bottleneck. In a permissioned setting with a static set of parties, it is common to proactively refresh the threshold setup [34]. In a proof-of-stake (PoS) setting, particularly, where the stake is constantly evolving and parties may dynamically join or leave the protocol, the ADKG protocol must be run periodically. Recent literature on asynchronous consensus uses committees, which contain only a subset of the parties, reducing the communication complexity of BA even further to $O(n \log n)$ at the cost of tolerating only $t < (1-\epsilon)n/3$ corruptions for any $\epsilon > 0$ [23, 49]. As the protocol run by the committee assumes an honest supermajority, this paradigm comes with one of two significant drawbacks. Either the sampled committee has to be very large, so that its maximal corruption remains below $n/3$ with overwhelming probability [60]. Otherwise, in order to keep the committee size small, the corruption level in the ground population must be assumed lower than $n/3$ by a considerable margin. Directly porting this idea to ADKG results in the same drawbacks. Finally, existing DKG protocols support only flat structures, where every party has the same weight and in total $t < n/3$ parties are corrupted. They do not readily work for a setting where every party holds a different share of the stake.

Seeds in PoS protocols. PoS-based ATOB protocols and blockchains require, apart from common coins, a second type of randomness, usually referred to as a *seed*. In PoS blockchains there is the notion of *accounts* with stake on them, of *roles*, such as "produce the 42-nd block", and of a *lottery*, through which accounts win the right to execute roles. This is typically [59, 82] implemented using a verifiable pseudo-random function (VRF) [122]: each account has a private key for a VRF and applies it to the role, producing a pseudorandom value.

If this value is above a threshold then the account wins the right to execute the role. However, for this approach to work the lottery needs as input not only a role but also a seed. Without it, a party can operate with several accounts and move all its stake to the luckiest account. By including a seed in the lottery, however, and by using the stake distribution from a point in time *before* the seed was unpredictable one can mitigate this attack [59].

In practice one can use a common-coin protocol to produce the seeds. We remark, however, that the two randomness-generation protocols have different requirements. A common-coin scheme does not have to be always unpredictable and agreed-upon, but only with some constant probability [131, 40]. It should, however, be efficient, as it is used in every agreement instance within the broadcast protocol. On the other hand, the seed-generation protocol must always be unpredictable and agreed upon, but it can be slow, as it is only run periodically (e.g., once per epoch).

Contributions. In this work we address all the aforementioned limitations of randomness generation for the first time. We present an asynchronous common-coin protocol that

- requires no trusted setup,
- supports optimal resilience $t < n/3$,
- employs small committees and is concretely efficient,
- directly supports the PoS setting and dynamic participation,
- is modular and can be generically used in any ATOB protocol.

Our methods. We are motivated by the question whether one can use the simple, practical, and efficient approach of getting common coins from threshold setup without running inefficient and complicated protocols whenever the stake has shifted. Building on the idea of Blum *et al.* [23], we rely on the fact that there already exists a functional ATOB: we generate the setup assuming that we already have the ATOB, and then use the generated setup to keep the ATOB running. To maintain practical efficiency the crucial step is to avoid using fully homomorphic encryption. We achieve this by generating weaker setups than Blum *et al.* [23], nonetheless still strong enough for the continued execution of the ATOB.

A crucial observation is that coins consumed by Byzantine agreement do not need to be perfect, i.e., always unpredictable and agreed upon [40, 131]. Hence, instead of generating a single, perfect threshold setup, we generate several candidate setups, such that some *constant* fraction of them are good. Many ADKG protocols can be seen as doing this as their first step, but their next step is to combine them into a single perfect setup. In order to be combinable, the setups must be of a particular form, and the committee that holds the setup must be *good* (that is, contain less than a threshold corruptions) except with negligible probability. As our setups are not combined and our committees only need to be good with a constant probability, our protocols are simpler and more efficient, and use smaller committees.

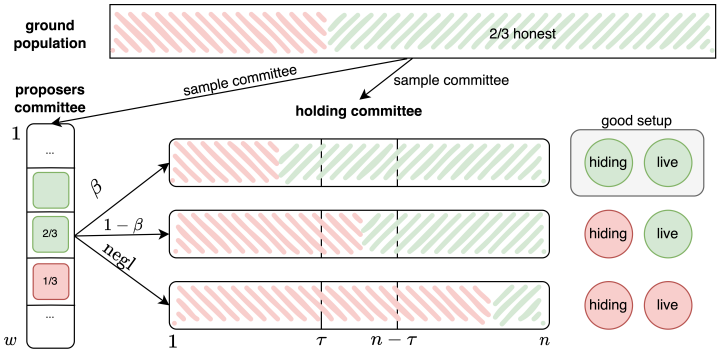


Figure 7.1. The high level idea of our protocols. A *proposers committee* is elected, and we wait until w proposers broadcast a setup. Assuming $2/3$ honesty in the ground population, a proposer is honest with probability $2/3$. Each proposer is assigned a *holding committee* of size n and creates an (n, τ) threshold setup for it. A committee is *hiding* if it contains at most τ corrupted parties, and *live* if it contains at most $n - \tau - 1$ corrupted parties. A setup is *good* if its proposer in honest and its holding committee is hiding and live. We set w so as to have enough honest proposers, and n and τ so that each holding committee is hiding with constant probability β and live with all but negligible probability. As a result, we get good setups with a constant probability γ.

Our common-coin protocol wMDCF follows the approach depicted in Figure 7.1. It elects a *proposers committee*, and each elected proposer is assigned a *holding committee*, for which it creates a threshold *coin*

setup. For this, the proposer acts as a dealer, encrypts the private setup material under the keys of the holding committee, and broadcasts these encryptions and the required verifications keys with a single message on the ATOB. We use a VRF-based lottery to determine both the proposers and the holding committees, where each party is elected with probability proportional to its stake. To flip a coin cid in the wMDCF protocol, we first hash cid with a seed to obtain a pointer to one of the published setups and then use that setup to obtain the value of cid. Which setup will be used for each cid is thus unpredictable until the seed is known.

Organization. The rest of this chapter is organized as follows. Section 7.3 presents the formal model used in the schemes and Section 7.4 presents the primitives used in our schemes. The common-coin protocol is presented in modular way, in two steps. First, Section 7.5 presents wHDCF, a weak honest-dealer coin-flip protocol, which is then used in Section 7.6 to build the wMDCF common-coin protocol. The schemes are parameterized over committee sizes and thresholds, and secure bounds for these are computed in Section 7.7. In Section 7.8 we analyze the concrete communication cost of our protocols.

7.2 Related work

Multiple common-coin constructions without a trusted dealer have been proposed in the literature. Ben-Or [20] presents a simple protocol, where every party flips a local coin. As a result, parties agree on the value of the coin only with probability $\Theta(2^{-n})$, A common-coin scheme from verifiable secret sharing has been shown by Canetti and Rabin [40], but their resulting Byzantine agreement protocol has communication complexity $\mathcal{O}(n^{11})$. Patra, Choudhury, and Rangan [131] bring this down to $\mathcal{O}(n^3)$.

A different approach constructs common coins from publicly verifiable secret sharing. The resulting protocols, such as SCRAPE [41], ALBATROSS [42], Spurt [56], HydRand [144], and RandHound-RandHerd [155], are efficient, yet they all make synchrony assumptions. RandShare [155] has been formalized in the asynchronous communication setting, but it is, according to its authors, less efficient.

Another line of work is based on time-based cryptography. Protocols in this category, such as Unicorn [104] and Bicorn [46], employ verifiable delay functions [25] and rely on the assumption that certain functions

(such as exponentiation in groups of unknown order [141]) can only be computed serially. None of the aforementioned works explicitly mentions the network assumptions. Overviews of random beacon protocols are given by Raikwar and Gligoroski [139], and by Choi, Manoj, and Bonneau [47].

Multiple works that circumvent ADKG exist in the literature, but they either make more assumptions, have non-optimal resilience, or result in inefficient protocols. Existing PoS blockchains rely on the timely delivery of honestly generated blocks, hence make timing assumptions. Ouroboros Praos [59] implements a randomness beacon protocol, used as seed in their leader-election algorithm, by hashing a large number of VRF outputs. Partial-synchrony assumptions assure that the honestly generated VRF outputs cannot be delayed arbitrarily by the adversary. King and Saia [100, 101] propose a synchronous common-coin protocol that makes uses of pseudorandomly selected committees, but achieves non-optimal resilience. This is improved in the protocol of Algorand [82, 45], where each committee member applies a VRF on the seed of previous block, and then the smallest valid VRF value sent by some committee member is kept. The protocol is first described in the synchronous model [82] and later extended to the partially synchronous [45]. Cohen, Keidar, and Spiegelman [49] extend this idea to the asynchronous model, but their protocol achieves an $n = 4.5t$ resilience. In all these protocols the coins are not reusable and the whole coin-generation algorithm has to be run repeatedly.

Blum *et al.* [23] also generate randomness without ADKG. Their ATOB protocol works in the following way. Assume first that a trusted dealer publishes on a ledger all the setup material required for one instance of Byzantine agreement and one instance of a multiparty computation (MPC) protocol. Then, on every invocation of the agreement protocol, parties use the Byzantine-agreement setup in the agreement protocol and the MPC setup in a tailor-made MPC protocol that refreshes the whole setup. Finally, they replace the trusted dealer with a standard MPC protocol, executed once in a distributed setup phase. This blueprint solves the problem of dynamic stake elegantly, but, the proposed MPC protocol for refreshing the setup, which has to be executed for *every* Byzantine agreement instance, is not efficient: it employs threshold fully homomorphic encryption, digital signatures, and zero-knowledge proofs.

7.3 Model

Ledger. We assume a model with asynchronous authenticated point-to-point channels. In addition, we assume an asynchronous persistent total-order broadcast channel. We denote by Ledger the totally-ordered sequence of messages that have been delivered on the channel. We point out that if a blockchain has a distinction between final and non-final messages, then Ledger denotes the final messages. We assume that when a protocol is started all the parties taking part in the protocol agree on a session identifier sid and an existing point on the ledger, $p \leq |\mathsf{Ledger}|$. We think of p as the *starting point of the protocol*, which gives consensus on the context of the protocol like stake distribution and lottery as discussed below. Protocols can have *public output* which might not be explicitly posted on the ledger, but will have a well-defined value and virtual point p at which they happened.

Definition 13 (Public output). We say that $\mathsf{PubOutF}$ is a public output function if it computes a public output from a ledger Ledger and a session identifier sid, where either $\mathsf{PubOutF}(\mathsf{Ledger}, \mathsf{sid}) = y \in \{0,1\}^*$ or $\mathsf{PubOutF}(\mathsf{Ledger}, \mathsf{sid}) = \perp$. We require that if $\mathsf{PubOutF}(\mathsf{Ledger}, \mathsf{sid}) \neq \perp$ then $\mathsf{PubOutF}(\mathsf{Ledger}\|m, \mathsf{sid}) = \mathsf{PubOutF}(\mathsf{Ledger}, \mathsf{sid})$ for all m. We say that sid gave public output y at position p if $|\mathsf{Ledger}| \geq p$ and $\mathsf{PubOutF}(\mathsf{Ledger}[1, p-1], \mathsf{sid}) = \perp$ and $\mathsf{PubOutF}(\mathsf{Ledger}[1, p], \mathsf{sid}) = y$. Unless multiple sid's are in scope we will omit the sid parameter. Finally we will informally say that some protocol gives public output $\mathsf{PubOutF}$ when additionally the ledger is implicit or when it is an eventual property of the ledger.

Dynamic stake. We consider proof-of-stake defined via the ledger. For each Ledger there is a stake distribution $\Sigma(\mathsf{Ledger}) : \mathbb{P} \to \mathbb{R}_0$ which may change as the ledger grows, can be computed in poly-time, and which gives for each party P its stake $\Sigma(\mathsf{Ledger})(\mathsf{P})$. For each point p there is also a stake distribution Σ_p, which is the stake distribution used by protocols with p as starting point. It may be different from $\Sigma(\mathsf{Ledger}[1, p])$, as discussed below.

Lotteries. In PoS based protocol it is common that parties are selected at random for carrying out a role in the protocol, like serving on a committee or producing the next block in a blockchain. To keep the model simple we assume that this is done via a random oracle. To keep

the model simple we assume that for each point p on the ledger there is a random oracle $\Gamma_p : \{0,1\}^* \to \{0,1\}^\lambda$. We assume that Γ_p is sampled and made available to the parties at some point *after* Σ_p can be computed from Ledger. This ensure that Γ_p is independent of Σ_p. If Γ_p was made available before Σ_p was fixed then corrupted parties could update Σ_p based on Γ_p and for instance give more stake to parties "lucky" in Γ_p. One way to implement this is to iteratively generate random and unpredictable seeds seed appearing as public outputs. Then for a given point p let seed_p be the latest seed on $\mathsf{Ledger}[1,p]$, let $\Gamma_p(x) = R(\mathsf{seed}_p, x)$ for a random oracle R, let $p' < p$ be the latest point where seed was unpredictable, and let $\Sigma_p = \Sigma(\mathsf{Ledger}[1,p'])$.

Our protocols include steps where a party samples a committee cid of size n. We model this as a function $\mathsf{SampleCommittee}_p(\mathsf{cid}, n) \to \left(\mathsf{H}_i\right)_{i\in[n]}$ that uses Γ_p to sample n parties from $\mathbb{P}$ with probability proportional to the stake Σ_p. As the input is public, the output can be verified by a function $\mathsf{VerifyCommittee}_p(\mathsf{cid}, \left(\mathsf{H}_i\right)_{i\in[n]})$ that reruns $\mathsf{SampleCommittee}_p(\mathsf{cid}, n)$ and verifies that it matches $\left(\mathsf{H}_i\right)_{i\in[n]}$. We assume $\mathsf{SampleCommittee}_p(\mathsf{cid}, n)$ is locally computable by every party. Using our lottery abstraction this could be implemented by calling $\Gamma_p(\mathsf{cid}, i)$, for some committee cid and for $i \in [n]$, to obtain a number $r_i \in \{0, 2^\lambda - 1\}$, and then deterministically mapping r_i to a party $P_i \in \mathbb{P}$ based on Σ_p. Observe that a party with relatively large stake can appear multiple times in the committee.

7.4 Primitives

Our schemes make use of the following primitives.

7.4.1 Public-key encryption with full decryption

There are keys $(\mathsf{dk}_i, \mathsf{ek}_i)$, for all $\mathsf{P}_i \in \mathbb{P}$, for an IND-CPA encryption scheme with full decryption, PKE. Encrypting a message $m \in \mathsf{PKE}.\mathcal{M}$ using randomness $r \in \mathsf{PKE}.\mathcal{R}$ results in a ciphertext $c = \mathsf{Enc}_{\mathsf{ek}_i}(m; r) \in \mathsf{PKE}.\mathcal{C}$. Given a ciphertext $c \in \mathsf{PKE}.\mathcal{C}$ the decryption algorithm $\mathsf{Dec}_{\mathsf{dk}_i}(c)$ returns both $m \in \mathsf{PKE}.\mathcal{M}$ and $r \in \mathsf{PKE}.\mathcal{R}$. The triple (m, r, c) can then be verified by anyone holding ek_i by checking if $\mathsf{Enc}_{\mathsf{ek}_i}(m; r) = c$.

Given an invalid ciphertext a zero knowledge proof that the ciphertext is invalid can be obtained using the secret key.

Construction using El Gamal. We first show that we can obtain the properties above in the random oracle model, as long as only encryptions of random messages are needed. This can then be lifted to a complete encryption scheme by symmetrically encrypting the message under a freshly sampled random key.

To encrypt a random value r, use El Gamal with $H(r)$ as randomness. I.e. if $\mathsf{dk} = x$ and $\mathsf{ek} = h = g^x$, then you encrypt r as $c = (A, B) = (g^{H(r)}, r \cdot h^{H(r)})$. To decrypt you first compute $r = B/(A^x)$, then check if re-encrypting using $H(r)$ as randomness gives back c. If verification checks out you can simply send r as proof. If the re-encryption does not match, you provide a proof that r was obtained by decrypting c. Note that (A, B) decrypts to r (under (g, h)) iff $\mathrm{DL}_g(h) = \mathrm{DL}_A(B/r)$, so this proof can be constructed using the Fiat-Shamir transform of the Σ-protocol for equality of discrete logarithms.

In the full scheme, in order to encrypt m using randomness r, you encrypt r as above and additionally include a symmetric encryption of m using r as key.

To decrypt you first use regular El Gamal decryption to obtain r and verify it by re-encrypting. If it was encrypted correctly you use it to decrypt m and return (m, r), otherwise return $(\perp, r)$.

7.4.2 Threshold coin flip

We use a (n, t)-threshold coin-flip (CF) scheme, where n is the total number of parties, t is the corruption threshold, and the reconstruction threshold is $t + 1$. The scheme has the following interface.

- $\mathsf{Setup}(n, t) \to (\mathsf{vk}, sk_1, \ldots, sk_n)$: The dealer generates a verification key vk and secret key shares sk_i of P_i. The secret keys can be used to create coin shares of multiple coins.
- $\mathsf{VerifyKeyShare}(\mathsf{vk}, i, sk_i) \to b \in \{0, 1\}$: Given the verification keys vk, it verifies sk_i.
- $\mathsf{Flip}(sk_i, \mathsf{coin}) \to (\mathsf{s}_i, \mathsf{w}_i)$: Given a coin identifier coin and secret key sk_i, it returns a coin share s_i for coin and potentially a correctness proof w_i, i.e., a proof that the coin share has been computed correctly using sk_i.
- $\mathsf{VerifyCoinShare}(\mathsf{vk}, \mathsf{coin}, \mathsf{s}_i, \mathsf{w}_i) \to b \in \{0, 1\}$: It verifies coin share s_i for coin identifier coin using the correctness proof w_i and verification key vk.

- $\mathsf{Combine}(\mathsf{coin}, \{\mathsf{s}_{i_j}\}_{j\in[t+1]}) \to \mathsf{s} \in \{0,1\}^\lambda$: Given $t+1$ valid coin shares s_{i_j}, for $j \in [t+1]$, it returns the value s of the coin identifier coin.
- $\mathsf{VerifyCoin}(\mathsf{vk}, \mathsf{coin}, \mathsf{s}) \to b \in \{0,1\}$: It verifies s as the value of coin identifier coin using the verification key vk.

Security properties. Assuming an honest dealer, i.e., that $\mathsf{Setup}()$ is correctly executed, and that there are no more than t corrupted parties, the scheme satisfies the following properties.

Completeness: If the dealer is honest then all key shares generated with $\mathsf{Flip}(sk_i, \mathsf{coin})$ will verify with $\mathsf{VerifyCoinShare}$.

Agreement: For any $t+1$ valid key shares the value $\mathsf{Combine}(\mathsf{coin}, \{\mathsf{s}_{i_j}\}_{j\in[t]})$ is the same, which define *the* value $\mathsf{s}_{\mathsf{coin}}$.

Unpredictability: The value $\mathsf{s}_{\mathsf{coin}}$ is unpredictable without honest shares, i.e., for a set $C = \{\mathsf{P}_{i_j}\}_{j\in[t+1]}$ of corrupted parties, if a poly-time adversary has been given vk and sk_i for $\mathsf{P}_i \in C$ for a random setup and has not been given $\mathsf{Flip}(sk_i, \mathsf{coin})$ for $\mathsf{P}_i \notin C$, then it cannot guess $\mathsf{s}_{\mathsf{coin}}$ better than at random. This holds even if it has access to an oracle giving $\mathsf{Flip}(sk_i, \mathsf{coin}')$ for all honest P_i for all $\mathsf{coin}' \neq \mathsf{coin}$.

Instantiation. Scheme CF can be instantiated with any non-interactive unique threshold signature scheme, such as BLS threshold signatures [27, 24]. The dealer picks a random secret key sk and shares it among all n parties using a polynomial $\phi(X) = \sum_{k=0}^{t} \phi_k X^k$, such that $\phi_0 = sk$. The only difference from threshold BLS is in $\mathsf{Setup}()$: it runs the key generation algorithm of the threshold signature scheme, but it does not return the verification keys in the form $g_2^{sk_i} \in G_2$, where $i \in [n]$ and g_2 is the generator of G_2, as in the original scheme. Instead, it returns a vector $(V_0, \ldots, V_t)$, where $V_k = g^{\phi_k} \in G_2$, for $k \in \{0, \ldots, t\}$, i.e., it returns Feldman commitments [74] to the coefficients of ϕ. This allows us to implement $\mathsf{VerifyKeyShare}()$, so P_i can verify that its key share sk_i is indeed a point on polynomial ϕ by checking whether

$$g_2^{sk_i} \stackrel{?}{=} \prod_{k=0}^{t} (V_k)^{i^k}. \tag{7.1}$$

Observe that the original verification keys can still be obtained using (7.1) with input vk and i, hence $\mathsf{VerifyCoinShare}()$ and $\mathsf{VerifyCoin}()$ need

no modification. Algorithm $\mathsf{Flip}()$ returns a signature share s_i on message coin using the key share sk_i of party P_i. Algorithm $\mathsf{Combine}()$ creates the threshold signature s from $t+1$ valid signature shares, which can then be hashed to get a value in $\{0,1\}$. Algorithms $\mathsf{VerifyCoinShare}()$ and $\mathsf{VerifyCoin}()$ invoke the signature verification algorithm, which, in the case of BLS, only takes as input the message coin and a signatures share s_i or signature s, i.e., $\mathsf{w}_i = \perp$, and uses a pairing function. Alternatively, one can use the common-coin scheme of Cachin, Kursawe, and Shoup [35], but $\mathsf{VerifyCoin}()$ would additionally need as input the $t+1$ valid coin shares and proofs $\{\mathsf{s}_{i_j}, \mathsf{w}_{i_j}\}_{j\in[t+1]}$.

7.4.3 Secret sharing

Our construction requires a secret sharing scheme TSS with threshold t with the following interface.

1. $\mathsf{Share}(s; r) \to (s_1, \ldots, s_n)$: It shares a secret s using randomness r to n secret shares $(s_1, \ldots, s_n)$.
2. $\mathsf{Reconstruct}(\{s_{i_j}\}_{j=1}^{t}) \to s'$: Given t shares it reconstructs some secret s'.

The hiding property says that the joint distribution of t shares s_i is independent of s. We can instantiate TSS with Shamir's secret sharing scheme [147].

7.4.4 Digital Signatures

Finally, there are keys $(sk_{\mathsf{P}}, \mathsf{vk}_{\mathsf{P}})$, for all $\mathsf{P} \in \mathbb{P}$, for a digital signature scheme DS with unique signatures.

7.4.5 Seed-generation protocol

Our common-coin scheme wMDCF uses a seed-generation sub-protocol seed. A seed can be thought of as a perfect coin flip: there is agreement on the output and its value is unpredictable before the protocol starts. The exact protocol we use is presented in the full version of this work [7]. It follows the same idea as our common-coin scheme: parties are pseudorandomly elected to form the proposers committee, and each proposer is assigned a holding committee, to which it secret-shares a random value. When a new seed value is needed, the members of each holding committee recombine the secret-shared value, and all the values

are then added together. The construction makes sure that, except with negligible probability, the protocol remains live and the value of the seed is unpredictable.

Syntax. The scheme has the following syntax.

Commit: On input (SEED, nid) in a session with session identifier nid a party starts running the commit protocol and may as a result public output PubOutSeedCommit.

Open: On input (SEED-OPEN, nid), which must be given after public output PubOutSeedCommit, in a session with id nid a party starts running the opening protocol and may as a result output (DONE-SEED, nid, c), for $c \in \{0,1\}^\lambda$.

Security properties. It satisfies the following security properties.

Termination: If all honest parties get input (SEED, nid) then eventually all honest parties give public output PubOutSeedCommit.

If all honest parties get correct input (SEED-OPEN, nid) then eventually all honest parties give an output (DONE-SEED, nid, $\cdot$).

Agreement: If two honest parties have outputs (DONE-SEED, nid, c_P) and (DONE-SEED, nid, c_Q), then $c_\mathsf{P} = c_\mathsf{Q}$. Call the common value c_nid.

Unpredictability: For each session nid it holds that c_nid is unpredictable before the first honest party gets input (SEED-OPEN, nid).

7.5 Weak honest-dealer coin flip

In this section we define the weak honest-dealer coin-flip (wHDCF) protocol. In wHDCF there is a designated dealer D, which is one of the participating parties. We assume D is given as part of session identifier, $\mathsf{sid} = (\mathsf{D}, \mathsf{sid}')$, and hence is known by all parties when the instance is created. The scheme is *weak* in the sense that parties may output $\perp$ as the value of the coin, but if two honest parties output a value in $\{0,1\}$, then it will be the same. It is *honest-dealer* as the coin value becomes predictable for a corrupted D. The scheme makes use of a committee verification mechanism $\mathsf{SampleCommittee}_p()$ proportional to stake at point p (Section 7.3), an encryption scheme with full decryption PKE (Section 7.4.1), and an (n_{COIN}, τ_{COIN})-threshold weak coin flip scheme CF

(Section 7.4.2). Here n_{COIN} and τ_{COIN} are protocol parameters, for which we choose specific values in Section 7.7.

Syntax. The syntax of weak honest-dealer coin-flip is as follows:

Deal**:** On input $(\text{DEAL}, \mathsf{sid})$ a participating party starts running the dealing protocol of CF and may as a result produce a public output $\mathsf{PubOutSingleDeal}$.

Flip**:** On input $(\text{FLIP}, \mathsf{sid}, \mathsf{cid})$, for coin identifier cid, after $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$, a party starts running the flip protocol of CF and outputs $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, s, \pi)$, where $s \in \{\perp\} \cup \{0,1\}^{\lambda}$ and π is a proof that s is the output of the coinflipping protocol. The proof can be checked by any party P' for which $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}_{\mathsf{P}'}) \neq \perp$ using $wHDCFVerify(\pi, m)$.

Security. The security properties of wHDCF are as follows.

Termination: (1) If D is honest and all honest parties get input $(\text{DEAL}, \mathsf{sid})$, then eventually $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$.

(2) If, after $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$, all honest parties get input $(\text{FLIP}, \mathsf{sid}, \mathsf{cid})$, then eventually all honest parties give output $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, \cdot)$, except with negligible probability.

Weak agreement: If two honest parties output $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{P}}, \pi)$ and $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{Q}}, \pi)$, such that $c_{\mathsf{P}} \neq \perp$ and $c_{\mathsf{Q}} \neq \perp$, then $c_{\mathsf{P}} = c_{\mathsf{Q}}$, except with negligible probability. Moreover, if D is honest, then no honesty party P outputs $c_{\mathsf{P}} = \perp$.

Honest-dealer β-unpredictability: If dealer D of session sid is honest, then each coin flip cid is independently unpredictable with some constant probability $\beta > 0$, where β is defined when $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$ and is independent of cid.

Construction. In a high level, the scheme works as follows. Dealer D is assigned a *coin-holding committee* of size n_{COIN} and creates a *coin setup* for an $(n_{\text{COIN}}, \tau_{\text{COIN}})$-threshold coin scheme CF for this committee. Termination is achieved by appropriately setting the parameters and from the pseudorandom nature of the committee: if the dealer completes the setup, there are at least $\tau_{\text{COIN}}+1$ honest parties in the committee, except with negligible probability. The weak agreement property is achieved by *verifiable* complaints against a corrupted dealer. Upon receiving a complaint valid complaint, a party terminates the Flip protocol outputting $\perp$.

If, additionally, D is honest, then our protocol guarantees unpredictability with constant probability β, defined as the probability of having at most τ_{COIN} corruptions in the committee, and depending only on n_{COIN} and τ_{COIN}.

Algorithm 9 Scheme wHDCF, algorithm Deal, where an instance sid of wHDCF is created at point p on Ledger. Code for process P_i.

161:**upon input** $(\text{DEAL}, \mathsf{sid})$ **where** $\mathsf{sid} = (\mathsf{D}, \mathsf{sid}')$ **and** $\mathsf{P}_i = \mathsf{D}$ **do** // only D
162: $(\mathsf{H}_1, \ldots, \mathsf{H}_{n_{\text{COIN}}}) \leftarrow \mathsf{SampleCommittee}_p(\mathsf{sid}, n_{\text{COIN}})$,
163: $(\mathsf{vk}, sk_1, \ldots, sk_{n_{\text{COIN}}}) \leftarrow \mathsf{CF.Setup}(n_{\text{COIN}}, \tau_{\text{COIN}})$
164: **for** $j \in [n_{\text{COIN}}]$ **do**
165: $r_j \xleftarrow{\$} \{0,1\}^\lambda$
166: $e_j = \mathsf{PKE.Enc}_{\mathsf{ek}_j}((sk_j, r_j))$
167: broadcast $(\mathsf{sid}, \mathsf{vk}, (\mathsf{H}_1, e_1), \ldots, (\mathsf{H}_{n_{\text{COIN}}}, e_{n_{\text{COIN}}}))$ on Ledger

In Algorithm 9 we implement Deal. The dealer first (line 162) samples the coin-holding committee of size n_{COIN} and then (line 163) uses CF to create a coin setup for it. The coin setup includes secret keys $sk_1, \ldots, sk_{n_{\text{COIN}}}$ and verification key vk. Each secret key sk_i is encrypted to party's P_i long term private key ek_i using a fresh randomness r_i (lines 164–166). The coin setup is broadcast on Ledger. When a coin-setup is included in Ledger we define the public output $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger})$ as $(\mathsf{vk}, (\mathsf{H}_1, e_1), \ldots, (\mathsf{H}_{n_{\text{COIN}}}, e_{n_{\text{COIN}}}))$ if the included committee verifies using $\mathsf{VerifyCommittee}$. Otherwise the output is $\perp$.

In Algorithm 10 we implement Flip. Only parties in the coin-holding committee run it. When P_i gets input $(\text{FLIP}, \mathsf{sid}, \mathsf{cid})$ and $\mathsf{PubOutSingleDeal}(\mathsf{sid}, \mathsf{Ledger}_{\mathsf{P}_i}) \neq \perp$, it first reads the coin setup and tries to decrypt e_i to obtain its key share (line 173). Scheme PKE returns sk_i' and the randomness r_i' that D is supposed to have used at encryption time. Party P_i checks whether D has indeed done so by re-encrypting (sk_i', r_i') and checking the result against e_i. If it is different, P_i sends a COMPLAINTENCRYPTION message that includes a zero-knowledge proof that e_i decrypts to (sk_i', r_i') (lines 174–177) and stops handling the Flip event. Otherwise, P_i can prove correct decryption of e_i in a complaint message by sending (sk_i', r_i'). Party P_i then verifies its key share against the verification vector vk published in the coin setup, and, if it is invalid, sends a COMPLAINTKEYSHARE message to $\mathcal{C}$ (lines 178–179) and returns. If the check passes, it creates a coin

Algorithm 10 Scheme wHDCF, algorithm Flip (cid), where an instance sid of wHDCF is created at point p on Ledger. Code for process P_i, P_i is one of the H_j in the coin-setup $(\mathsf{sid}, \mathsf{vk}, (\mathsf{H}_1, e_1), \ldots, (\mathsf{H}_{n_{\text{COIN}}}, e_{n_{\text{COIN}}}))$ published on Ledger.

State:

validShares[sid][cid] $\leftarrow [\,]$; terminated[sid][cid] $\leftarrow 0$

justifiedComplaint[sid][cid] $\leftarrow \perp$, for each sid and cid

upon input (FLIP, sid, cid) **such that** PubOutSingleDeal(sid, Ledger) $\neq \perp$ **do**
  $(\mathsf{vk}, (\mathsf{H}_1, e_1), \ldots, (\mathsf{H}_{n_{\text{COIN}}}, e_{n_{\text{COIN}}})) \leftarrow$ PubOutSingleDeal(sid, Ledger)
  let $\mathcal{C} = \{\mathsf{H}_1, \ldots, \mathsf{H}_{n_{\text{COIN}}}\}$
  $(sk_i', r_i') = \mathsf{PKE.Dec}_{\mathsf{dk}_i}(e_i)$
  if $e_i \neq \mathsf{PKE.Enc}_{\mathsf{ek}_i}((sk_i', r_i'))$ **then**
    create zk-proof W_i that e_i decrypts to (sk_i', r_i')
    send (COMPLAINTENCRYPTION, sid, cid, $\mathsf{W}_i, sk_i', r_i'$) to parties in $\mathcal{C}$
    return
  if CF.VerifyKeyShare(vk, i, sk_i') $= 0$ **then**
    send (COMPLAINTKEYSHARE, sid, cid, sk_i', r_i') to parties in $\mathcal{C}$; **return**
  $\mathsf{s}_i = \mathsf{CF.CreateShare}(sk_i', \mathsf{cid})$
  send (COINSHARE, sid, cid, s_i) to parties in $\mathcal{C}$

upon deliver (COINSHARE, sid, cid, s_j) from P_j **do**
  if CF.VerifyCoinShare(vk, cid, s_j) $= 1$ **then**
    append s_j to validShares[sid][cid]

upon deliver $c = $ (COMPLAINTENCRYPTION, sid, cid, W_j, sk_j, r_j) **do**
  $e_j' \leftarrow \mathsf{PKE.Enc}_{\mathsf{ek}_j}((sk_j, r_j))$
  if $e_j' \neq e_j$ and W_j is valid **then**
    justifiedComplaint[sid][cid] $\leftarrow c$

upon deliver $c = $ (COMPLAINTKEYSHARE, sid, cid, sk_j, r_j) **do**
  $e_j' \leftarrow \mathsf{PKE.Enc}_{\mathsf{ek}_j}((sk_j, r_j))$
  if $e_j' = e_j$ **and** CF.VerifyKeyShare(vk, i, sk_i) $= 0$ **then**
    justifiedComplaint[sid][cid] $\leftarrow c$

upon |validShares[sid][cid]| $= \tau_{\text{COIN}} + 1$ **and** terminated[sid][cid] $= 0$ **do**
  let $(s_{j_1}, \ldots, s_{j_{\tau_{\text{COIN}}+1}}) =$ validShares[sid][cid]
  $s \leftarrow \mathsf{CF.Combine}(\mathsf{cid}, \{s_{j_k}\}_{k \in [\tau_{\text{COIN}}+1]})$
  terminated[sid][cid] $\leftarrow 1$
  output (DONE-FLIP, sid, cid, s, validShares[sid][cid])

upon justifiedComplaint[sid][cid] $\neq \perp$ **and** terminated[sid][cid] $= 0$ **do**
  terminated[sid][cid] $\leftarrow 1$
  output (DONE-FLIP, sid, cid, $\perp$, justifiedComplaint[sid][cid])

share using the threshold-coin scheme CF (line 180) and sends to the committee $\mathcal{C}$. All complaints are verifiable: COMPLAINTENCRYPTION is valid if the zk-proof W_j, proving that the published e_j decrypts to (sk_j, r_j)), is valid, and the re-encryption of (sk_j, r_j) gives something different from e_j (lines 185–188). COMPLAINTSHARE is valid if the re-encryption of (sk_j, r_j) gives the published e_j and the key share sk_j is deemed invalid by the CF scheme. (lines 189–192). Party P_i outputs in two cases, whichever comes first. First, upon collecting $\tau_{\text{COIN}} + 1$ valid coin shares (line 193), in which case the value of the coin is reconstructed using the underlying CF scheme. Second, upon receiving a valid complaint (line 198), in which case a $\perp$ value is output.

The *wHDCFVerify* check can be implemented by a function that verifies a complaint according to the activation rules in line 185 or line 189, or, when given a set of shares, checks that they are valid and reruns the activation rule in line 193.

Proofs. We first present a more formal version of *Honest-dealer β-unpredictability* and then show the proofs. In the following, let f denote the actual number of corrupted parties in the coin-holding committee. Then, denote by GOOD-SETUP, NO-UNPRED, NO-LIVE the events that $f \in [0, \tau_{\text{COIN}}]$, $f \in (\tau_{\text{COIN}}, n_{\text{COIN}} - \tau_{\text{COIN}})$, and $f \in [n_{\text{COIN}} - \tau_{\text{COIN}}, n_{\text{COIN}}]$, respectively. As the names suggest, all protocol properties will be satisfied in the first case, while unpredictability may be violated in the second, and additionally liveness may be violated in the third. In Section 7.7 we choose concrete parameters for the committee election, such that $\Pr[\text{GOOD-SETUP}] = \beta$, for β a constant, and $\Pr[\text{NO-LIVE}]$ is negligible. Observe that $\mathsf{SampleCommittee}_p()$ is verifiable and unpredictable (see Section 7.3), hence D cannot affect the probability of these events.

Definition 14 (Honest-dealer β-unpredictability for wHDCF). We formalize honest-dealer unpredictability as a challenge game played against the PPT adversary $\mathcal{A}$. The definition implicitly assumes the termination property, as it does not make sense to define unpredictability of a value which is not defined, and the agreement property, as it is trivial to guess one of the outcomes if two honest parties have different coins. The following definition demands that the dealing phase results in a "good" setup with a constant probability, and, if D is honest, such a good setup always leads to unpredictable coin flips.

Sometimes good: For all sid there exists an event $\mathsf{Bad}_{\mathsf{sid}}$, which can be defined in PPT from the view of the adversary once the first honest

party gives output (DONE-DEAL, sid), i.e., the adversary knows if the event happened. It should hold that $\mathsf{Bad}_{\mathsf{sid}}$ happens with probability at most $1-\beta$, independently for each sid and independently from D being honest.

Unpredictable when good: Furthermore, the adversary can win the following game with probability at most negligibly better than $2^{-\lambda}$. The adversary may initiate as many sessions sid and coin-flips cid as it wants. At some point it specifies $(\mathsf{sid}, \mathsf{cid}, g)$, where sid is a session with an honest dealer, where $\mathsf{Bad}_{\mathsf{sid}}$ did not happen, where no honest party got input (FLIP, cid) yet, and where $g \in \{0,1\}^\lambda$. The adversary wins if it can execute to the point where some honest party outputs (DONE-FLIP, cid, c) with $c = g$.

Termination. (1) For the first part, assume D is honest. Since D does not wait for any parties in any step, it successfully broadcasts the coin setup, and, from the liveness property of Ledger, it will eventually be delivered on Ledger. (2) Assume (DONE-DEAL, sid) has been observed by all honest parties. Hence, honest parties can read the coin setup from Ledger and verify it using $\mathsf{VerifyCommittee}()$. By nature of committee election and by the choice of n_{COIN} and τ_{COIN} there are at least $\tau_{\text{COIN}} + 1$ honest coin holders. We distinguish two cases. Either the dealer has created valid key shares and encryptions for *all* honest coin holders, or there is *at least one* honest coin holder, for whom the dealer has created an invalid key share. Observe that the two cases cover all possible executions. If we are in the first case, then every honest coin holder P_i will successfully decrypt e_i to get a valid sk_i. From *termination* property of CF, and since there are at least $\tau_{\text{COIN}}+1$ honest coin holders, it follows that the coin shares of these parties are sufficient to reconstruct the coin value, hence eventually every honest party in the committee will output (FLIP, $\mathsf{sid}, \mathsf{cid}, \cdot$). If we are in the second case, the honest party can always compute a valid complaint against the dealer. If e_i is incorrect, P_i can prove this by broadcasting a complaint COMPLAINTENCRYPTION, which contains a zk-proof W_i that e_i decrypts to (sk_i', r_i'). A verifier can always verify the zk-proof and check that re-encrypting (sk_i', r_i') does not give e_i. If, on the other hand, e_i is a correct encryption of an invalid key share sk_i', P_i can also prove this broadcasting sk_i' and r_i' in a COMPLAINTSHARE message. A verifier can now verify that (sk_i', r_i') indeed re-encrypts to e_i, but $\mathsf{CF.VerifyKeyShare}(\mathsf{vk}, i, sk_i')$ returns 0. Honest parties will eventually deliver the complaint and output (FLIP, $\mathsf{sid}, \mathsf{cid}, \perp$). □

Weak agreement. Let P and Q be parties that output $c_\mathsf{P} \neq \perp$ and $c_\mathsf{Q} \neq \perp$ as the value of the coin using sets of $t+1$ valid coin shares S_P and S_Q in $\mathsf{CF.Combine}()$, respectively. Then, since the shares in S_P and S_Q are valid, they all lie on the same polynomial, hence they define the same secret, and $c_\mathsf{P} = c_\mathsf{Q}$. Additionally, if D is honest, then no valid complaint can be computed, except with negligible probability, hence honest parties P and Q output $c_\mathsf{P} \neq \perp$ and $c_\mathsf{Q} \neq \perp$, and, from the *agreement* property of CF, we get $c_\mathsf{P} = c_\mathsf{Q}$. □

Honest-dealer β-unpredictability. Define $\beta = \Pr[\text{GOOD-SETUP}]$ and $\mathsf{Bad}_\mathsf{sid} = \neg\text{GOOD-SETUP}$, i.e., $\mathsf{Bad}_\mathsf{sid}$ is the event that the committee assigned to D contains more than τ_{COIN} corrupted parties. Given sid and a point p on Ledger, the committee returned by $\mathsf{SampleCommittee}_p()$ is deterministic, hence $\mathsf{Bad}_\mathsf{sid}$ happens with probability $1-\beta$, which is constant and independent of sid, and can be defined from the view of the adversary once the coin setup appears on Ledger. The exact value of β depends on the choice of n_{COIN} and τ_{COIN}. The *Sometimes good* property is satisfied. Now about the *Unpredictable when good* property. When $\mathsf{Bad}_\mathsf{sid}$ does not happen, the number of actual corrupted coin holders is not more than τ_{COIN}. The *Unpredictability* property of the CF scheme holds in this case, and the *Unpredictable when good* property of wHDCF can be reduced to the *Unpredictability* of CF. □

Remark 5 (Weak agreement vs. honest-dealer aggreement). One can also aim for an *honest-dealer agreement* property, where, if D is honest, then honest parties output the same coin value. Our *weak agreement* property is stronger: if D misbehaves, then some parties may output $\perp$, but honest parties will never output different coin values. It reduces to *honest-dealer agreement* by having parties flip a local coin whenever $\perp$ is output.

7.6 Weak multiple-dealer coin flip

In this section we define the weak multiple-dealer coin-flip (wMDCF) protocol. It is *weak* as it inherits the agreement property from wHDCF: parties may output $\perp$, but if two honest parties output a value in $\{0,1\}$, then it will be the same. It is called *multiple-dealer* as there are multiple dealers, forming a *proposers committee*, selected pseudorandomly using $\mathsf{SampleCommittee}_p()$. The protocol uses parameters m_wMDCF and

w_{wMDCF}. Parameter m_{wMDCF} refers to the size of the proposers committee, i.e., the number of parties that are selected to act as a dealer in an instance of wMDCF. Parameter w_{wMDCF} refers to the number of parties in the proposers committee we asynchronously wait for. In Section 7.7 we show how to set these parameters, such that at least one good setup appears on the ledger, except with negligible probability, and a constant rate γ of the setups are good. The protocol makes use of seed, a seed-generation sub-protocol described in Section 7.4.5.

Syntax. The syntax of weak Multiple-Dealer Coin-Flip (wMDCF) is as follows:

Deal**:** On input $(\text{DEAL}, \mathsf{sid})$ a participating party starts running the dealing protocol and may as a result help produce a public output PubOutMultiDeal.

Flip**:** On input $(\text{FLIP}, \mathsf{sid}, \mathsf{cid})$ for a coin identifier cid, after $\mathsf{PubOutMultiDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$, a party starts running the coin-flip protocol and outputs $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, s)$, where $s \in \{\perp\} \cup \{0,1\}^{\lambda}$.

Security. The security properties of honest-dealer coin-flip are as follows. For the *agreement* and *unpredictability* properties we use a probability $\gamma > 0$, called the *good-setup probability*, which depends on the parameter w_{wMDCF} and on the hiding probability β of wHDCF, and is constant and independent of sid and cid.

Termination: (1)If all honest parties get input $(\text{DEAL}, \mathsf{sid})$ then eventually there is public output $\mathsf{PubOutMultiDeal}(\mathsf{sid}, \mathsf{Ledger}) \neq \perp$, except with negligible probability.

(2) If all honest parties get input $(\text{FLIP}, \mathsf{sid}, \mathsf{cid})$ then eventually all honest parties give an output $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, \cdot)$, except with negligible probability.

γ-Agreement: For each session sid and coin identifier cid it holds that, if two honest parties output $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{P}})$ and $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{Q}})$, such that $c_{\mathsf{P}} \neq \perp$ and $c_{\mathsf{Q}} \neq \perp$, then $c_{\mathsf{P}} = c_{\mathsf{Q}}$, except with negligible probability. Moreover, with probability γ it holds that no honest party outputs $\perp$ as the value of the coin. All together, this means that, if two honest parties have outputs $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{P}})$ and $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{Q}})$, then $c_{\mathsf{P}} = c_{\mathsf{Q}} \neq \perp$ with probability γ.

γ-Unpredictability: For each session sid and coin identifier cid it holds that the value of coin cid is unpredictable with probability γ.

Algorithm 11 Scheme wMDCF, algorithm Deal, where an instance sid of wMDCF is created at some point p on Ledger. Code for process P_i.

State:

201: setups$[w_{\mathsf{wMDCF}}] \leftarrow [\,]$

202:**upon input** (DEAL, sid) **do**
203: $\mathcal{C} \leftarrow \mathsf{SampleCommittee}(\mathsf{nid}, m_{\mathsf{wMDCF}})$
204: **for** $j \in [m_{\mathsf{wMDCF}}]$ **such that** $\mathcal{C}[j] = P_i$ **do**
205: wHDCF(Deal, $((\mathsf{P}_i, j), \mathsf{sid})$)

206:**upon** w_{wMDCF} setups PubOutMultiDeal$(((\mathsf{P}_j, k), \mathsf{sid}), \mathsf{Ledger}) \neq \perp$
207: **such that** $\mathcal{C}[k] = \mathsf{P}_j$ **do**
208: let setups contain the identifiers which gave public output
209: seed(SEED, sid)

Construction. On Algorithm 11 we implement Deal. On input (DEAL, sid), a protocol instance is created with some starting point p. For each time P_i is sampled to be a dealer in a wHDCF instance (line 203), it creates a new instance of wHDCF and runs the Deal algorithm. Every party waits for w_{wMDCF} instances of the wHDCF protocol (started by the dealers sampled in line 203) to give public output on the Ledger. When this happens, parties run an instance of the seed protocol (line 209). This seed will be later used in the Flip algorithm of wMDCF to pseudorandomly choose one of the w_{wMDCF} setups. We define PubOutMultiDeal = PubOutSeedOpen, so the output of the seed protocol signals the end of the dealing phase.

In Algorithm 12 we implement Flip. On input (FLIP, sid, cid) and after observing public output PubOutMultiDeal every party P_i uses a cryptographic hash function H, to hash (sid, cid, seed) into $j \in \{1, \ldots, w_{\mathsf{wMDCF}}\}$ (line 211). Then, the algorithm Flip of the wHDCF_j instance is used to compute the value of coin cid. We assume that each party on the committee of the selected wHDCF instance disseminate the output to the ground population.

Algorithm 12 Scheme wMDCF, algorithm Flip (cid), where an instance sid of wMDCF is created at some point p on Ledger. Code for P_i.

```
210:upon input (FLIP, sid, cid) such that PubOutMultiDeal(sid, Ledger) ≠ ⊥ do
211:     j ← H(sid, cid, seed)
212:     wHDCF(FLIP, setups[j], cid)

213:upon deliver (DONE-FLIP, sid, cid, s, π)
214:     if wHDCFVerify(π, s) then
215:          output (DONE-FLIP, sid, cid, s)
```

Proofs. We first present the *sometimes good* property and formalize the *agreement* and *unpredictability* properties, and then show the proofs.

Definition 15 (The *sometimes good* property). For all sid and cid there exists an event $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$, which is defined in PPT from the view of the adversary once the first honest party gives output (DONE-DEAL, sid), i.e., the adversary knows if the event happened. Moreover, there exists a probability $\gamma > 0$, called the *good-setup probability*, which depends on the parameters m_{wMDCF} and w_{wMDCF} and on the hiding probability β of wHDCF, is constant and independent of sid and cid, and $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$ happens with probability $1 - \gamma$.

We formalize agreement as a challenge game played against the PPT adversary $\mathcal{A}$. It assumes the termination property, so that the value of coin-flips actually become known.

Definition 16 (γ-Agreement). We require that the *sometimes good* property holds for $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$. Moreover, The adversary can win the following game with at most negligibly probability. The adversary may initiate as many sessions sid and coin flips cid as it wants. At some point it specifies $(\mathsf{sid}, \mathsf{cid}, \mathsf{P}, \mathsf{Q})$, where P has output $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{P}})$, Q has output $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c_{\mathsf{Q}})$, and sid and cid are such that $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$ did not happen. The adversary wins if $c_{\mathsf{P}} \neq c_{\mathsf{Q}}$.

We formalize unpredictability as a challenge game played against the PPT adversary $\mathcal{A}$. As with wHDCF, the definition assumes the termination and agreement properties.

Definition 17 (γ-Unpredictability). We require that the *sometimes good* property holds for $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$. Moreover, the adversary can win

the following game with probability at most negligibly better than $1/2$. The adversary may initiate as many sessions sid and coin-flips cid as it wants. At some point it specifies $(\mathsf{sid}, \mathsf{cid}, g)$, where $(\mathsf{sid}, \mathsf{cid})$ is such that $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$ did not happen, where no honest party got input $(\text{FLIP}, \mathsf{sid}, \mathsf{cid})$ yet, and where $g \in \{0, 1\}$. The adversary wins if it can execute to the point where some honest party outputs $(\text{DONE-FLIP}, \mathsf{sid}, \mathsf{cid}, c)$ with $c = g$.

Termination. (1) The dealing protocol waits in two places, first for w_{wMDCF} instances of wHDCF, and then for an instance of SEED. For the first one, the choice of parameters guarantees that, except with negligible probability, at least w_{wMDCF} instances of wHDCF will have an honest dealer, and each of these instances will terminate, according to the *termination* property of wHDCF. For the second, termination follows directly from the SEED protocol.

(2) As the Flip algorithm of wMDCF calls one of the wHDCF instances, the Flip termination of wMDCF directly follows from the Flip termination property of wHDCF. □

Sometimes good. Let $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$ be the event that $\mathsf{Hash}(\mathsf{sid}, \mathsf{cid}, \mathsf{seed})$ points to a *bad coin setup*, i.e., to a wHDCF instance sid_j whose dealer is corrupted or whose coin-holding committee has more than τ_{COIN} corruptions (which happens with probability β). The value of seed, which is included as a parameter when hashing to obtain sid_j, is unpredictable by the adversary, by the unpredictability property of seed, and becomes known *after* the w_{wMDCF} coin setups have appeared on Ledger. Hence, the probability of $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$ happening is independent of cid, and only depends on the probability β of $\mathsf{Hash}(\mathsf{sid}, \mathsf{cid}, \mathsf{seed})$ hitting a bad committee. Hence, this event happens with probability $1 - \gamma$, which is constant and independent of sid, cid. □

γ-Agreement. Let $(\mathsf{sid}, \mathsf{cid})$ such that $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$ did not happen, and hence the dealer of the sid_j CF instance, for $j = \mathsf{Hash}(\mathsf{sid}, \mathsf{cid}, \mathsf{seed})$, is honest. The property then follows from the *weak agreement* property of wHDCF. □

γ-unpredictability. Let $(\mathsf{sid}, \mathsf{cid})$ such that $\mathsf{Bad}_{\mathsf{sid},\mathsf{cid}}$ did not happen. This means that the dealer of the sid_j CF instance, for $j = \mathsf{Hash}(\mathsf{sid}, \mathsf{cid}, \mathsf{seed})$, is honest. It also implies that the event $\mathsf{Bad}_{\mathsf{sid}}$, defined in the proof *honest dealer β-unpredictability* property of wHDCF as the event that the committee of the sid_j instance contains more than τ_{COIN} corruptions, did not

happen. Hence, the property reduces to *Honest-dealer β-unpredictability* of wHDCF. □

7.7 Setting the parameters

Definition 18 (Binomial distribution). Let X a random variable counting the number of successes out of n trials, where success happens with probability p. Then X follows the binomial distribution, i.e., $X \sim \mathcal{B}(n, p)$ and the probability that exactly k successes happen is

$$\Pr[X = k] = \Pr[\mathcal{B}(n, p) = k] = \binom{n}{k} p^k (1-p)^{(n-k)}. \tag{7.2}$$

7.7.1 Sampling a holding committee for wHDCF

Let n denote the size of a holding committee and $\tau < n/2$ denote a number, such that the holding committee has at most τ corruptions with a constant probability β, and more than $n - \tau$ corruptions only with a negligible probability $\epsilon = 2^{-\lambda}$, where λ is the security parameter. The idea is the following. If we use a (n, τ)-secet-sharing or common-coin scheme in the committee, then the committee is *hiding* with probability β and *live* with probability $1 - \epsilon$. These capture the parameters of the wHDCF scheme, where we set $n \triangleq n_{\text{COIN}}$ and $\tau \triangleq \tau_{\text{COIN}}$.

As discussed earlier, we model a committee-election mechanism as a black-box function SampleCommittee(), which samples parties with probability proportional to their stake at some well-defined point on the ledger. In practice, this can be achieved by replacing each party with a (usually very large) number of smaller, atomic sub-parties, proportional to each party's stake, and use a VRF to pseudorandomly choose a sub-party [82]. In this section we assume that the ground population (the number of sub-parties) is very large, so that the probability of choosing a corrupted party does not change after choosing a party. Hence, SampleCommittee() does sampling with replacement, which can be modelled with a binomial distribution.

Using (7.2) we have that

$$\beta = \sum_{k=0}^{\tau} \Pr[\mathcal{B}(n, 1-p) = k]$$

and

$$\epsilon = \sum_{k=n-\tau+1}^{n} \Pr[\mathcal{B}(n, 1-p) = k],$$

for $p = 2/3$. In Table 7.1 we show various combinations for n and τ, such that $\epsilon \leq 2^{-\lambda}$ for $\lambda = 60$, and the resulting hiding probability β.

7.7.2 Sampling a proposer committee for wMDCF

In protocol wMDCF parties have a chance to participate in the *proposers committee*, i.e., to win the right to become a dealer in a wVSS or wHDCF instance, respectively. Parties are again sampled using SampleCommittee (Section 7.7.1), which returns a committee of size m, but the protocols only wait for the first w setups to appear on Ledger and only use those.

Necessary conditions. As before, we need to make sure that, except with negligible probability $\epsilon = 2^{-\lambda}$, there are at least w honest parties on the committee to ensure termination. This is bounded as ϵ in Section 7.7.1 but with n and τ replaced by m and $w-1$ respectively. But now we additionally need to make sure that, except with negligible probability at least one of the w setups that appear on Ledger is a *good setup*, that is, from an honest party who sampled a committee with less than τ corruptions. This condition corresponds exactly to the setup in Section 7.7.1 being hiding, but with the probability p changed to account for the fact that we are interested in the probability of not just an honest party but an honest party *who provided a good setup* making it into any subset of size w. Since an honest dealer has a β (which depends on the parameters of the subprotocol) probability of providing a bad setup, we set $p = \beta \cdot 2/3$ and require $\sum_{k=0}^{w-1} \Pr[\mathcal{B}(m, 1-p) = k] \geq 1 - 2^{-\lambda}$.

Good-setup probability. Finally, we calculate the probability γ, defined in Section 7.6, that a setup for wMDCF published on Ledger is good, i.e., the probability of getting an unpredictable and agreed upon value in each coin flip. We derive this from the expected number of bad setups, which (by linearity of expectation) is $m \cdot (1 - \beta \cdot \frac{2}{3})$, and from the fact that the adversary can schedule the order of messages, causing all bad setups and, hence, only $w - m \cdot (1 - \beta \cdot \frac{2}{3}))$ good setups, to appear on Ledger. This gives us the fraction of good setups that in expectation

appear on the ledger as

$$\gamma = \frac{w - (m \cdot (1 - \beta \cdot \frac{2}{3}))}{w}. \tag{7.3}$$

Putting it all together. We show the resulting parameters with $\lambda = 60$ bits of security in Table 7.1. As an example, for a holding committee with size $n = 259$ and reconstruction threshold $\tau = 103$, we get hiding probability $\beta = 98.7\%$. Then we can sample a proposers committee of size $m = 653$ and wait for $w = 327$. This results in $84,693$ encrypted shares being posted on the Ledger and for wMDCF it gives a good-setup probability $\gamma = 31.8\%$.

n	τ	β	m	w	γ	$n \cdot w$
653	320	$> 1 - 2^{60}$	653	321	32.2%	$209.6K$
300	125	99.9%	653	322	32.3%	$96.6K$
280	114	99.6%	653	323	32.1%	$90.4K$
275	111	99.4%	653	324	32.0%	$89.1K$
271	109	99.3%	653	325	32.0%	$88.1K$
265	106	99.0%	653	326	31.9%	$86.4K$
261	104	98.8%	653	327	31.9%	$85.3K$
259	103	98.7%	653	327	31.8%	$84.7K$
257	102	98.6%	659	330	31.6%	$84.8K$
256	101	98.3%	672	337	31.3%	$86.3K$
254	100	98.1%	682	342	31.1%	$86.9K$
252	99	98.0%	692	347	30.8%	$87.4K$

Table 7.1. This table shows possible values (subject to conditions in Section 7.7.1) for the *holding committee* parameters, n and τ, and the resulting hiding probability β. For each obtainable β, it shows possible values (subject to conditions in Section 7.7.2) for the *proposers committee* parameters, m and w, and the resulting good-setup probability γ. Each of the w dealers encrypts keys for a committee of size n, which 1 gives a total of $m * w$ encryptions.

7.8 Analysis of communication complexity

To demonstrate the power of being able to sample concretely small committees, we analyze the concrete complexity of our protocols. Note that

a purely asymptotic analysis would not show any gains over simply using a state of the art ADKG protocol with subset sampling and near optimal resilience. We give all sizes in *bits*, but for simplicity we treat group and field elements as λ bits. For instance, we use 3λ as the size of an encrypted share, which (using Section 7.4.1) consists of 2 group elements and a symmetrically encrypted share of a secret of size λ. This will not be true for concrete instantiations, but it only changes our estimates by a small constant factor which depends on e.g. the concrete curves being employed.

We define ATOB complexity as the cost of including a message of a given size in Ledger. In the following "broadcast" refers to broadcasting through the ATOB and "multicast" refers to a party sending a message to all parties in the ground population. As the communication cost of a broadcast and multicast depend on the concrete implementation we keep these costs opaque and report the results as a number of broadcasts and multicasts of various sizes. For inter-committee communication we assume point-to-point channels are used and give the results in total number of bits sent though the channels.

The wHDCF protocol has an ATOB complexity of 1 message of size $n_{\text{COIN}} \cdot 3\lambda + \lambda$ and no additional communication in the initial setup phase. The message complexity of each coin flip is at most ${n_{\text{COIN}}}^2 \cdot 4\lambda$ to reconstruct the coin (or $\perp$) in the committee, and then to disseminate the value to the ground population each committee member multicasts the reconstructed coin or a complaint of size at most 4λ, resulting in at most ${n_{\text{COIN}}}^2 \cdot 4\lambda$ bits communicated in addition to n_{COIN} multicasts of size 4λ.

The deal phase of the wMDCF protocol has the same complexity as m_{wMDCF} deal phases of wHDCF and a single run of the seed protocol. That is, an ATOB complexity of m_{wMDCF} messages of size $n_{\text{COIN}} \cdot 3\lambda + \lambda$ and m_{SEED} messages of size $n_{\text{VSS}} \cdot 3\lambda + \lambda$, a multicast complexity of $w_{\text{SEED}} \cdot n_{\text{VSS}}$ multicasts of size at most $(\tau_{\text{VSS}} + 1) \cdot 2\lambda$, in addition to a communication complexity of $m_{\text{SEED}} \cdot {n_{\text{VSS}}}^2 \cdot 4\lambda$ bits. Whenever a coin needs to be flipped using wMDCF, the message complexity is that of running the selected wHDCF protcol.

To refresh the setup after the stake distribution has changed, one would need to first run an instance of the seed protocol and then the deal phase of the wMDCF protocol. Using the best parameters in Table 7.1 the concrete cost of refreshing the setup is 1959 messages of size 778λ, and 169386 multicasts of size at most 208λ. The communication complexity of flipping a coin and disseminating it to all parties is $259^2 \cdot 4\lambda$ in addition to 259 multicasts of 4λ bits.

If we were to assume $t < 0.3n$ in the paradigm of "subset sampling with almost optimal resilience" [23], and need a committee with honest supermajority with probability $1 - 2^{-60}$, then one would need to sample a committee with 16037 parties [60, Table 1]. If we then instantiate a state of the art ADKG protocol with $O(n^3\lambda)$ communication using the committee and for the sake of an example assume the concrete cost is $n^3\lambda$, then we get a complexity of $> 4 \cdot 10^{12}\lambda$. It is then clear that our approach is far cheaper for all but extremely large values of n.

7.9 Discussion

In this work we have presented protocols for generating randomness in an asynchronous PoS setting with dynamic participation. The protocols are practical and concretely efficient, they employ no trusted setup, and they make use of small committees. We have computed concrete numbers for the committee size. Specifically, we can have a committee of $m = 653$ proposers, each generating a setup for $n = 359$ holders, resulting in approx. $85K$ encrypted values posted on Ledger. For $\kappa = 60$ bit of security and assuming optimal corruption $1/3$ in the ground population, this gives randomness-generation protocols that are live with all but negligible probability. Our common-coin protocol is unpredictable and agreed-upon with probability approx. 31.8%, and, as it is based on threshold cryptography, the setup can be used for a flipping a polynomial number of coins. These committee sizes result from the fact that we require not all but only a constant factor of our setups to be good.

It is instructive to compare these results against previous literature, particularly against the approach that runs the randomness-generation protocols in committees with honest supermajority. Algorand [82, Figure 3] requires a committee of size approx. 2000, assuming corruption 0.2 in the ground population, and larger than 4000, assuming corruption 0.24, to get good committees with probability $5 \cdot 10^{-9}$, or approx. 28 bits of security. Extending this approach to a ground population with corruption 0.3, which is still sub-optimal, and 60 bits of security, the authors of GearBox [60, Table 1] show that committees of size 16037 are needed. We remark that asynchronous distributed key generation protocols, the state-of-the-art approach for threshold-setup generation, require honest supermajority, hence one would require a committee of similar sizes and sub-optimal resilience in the ground population.

Chapter 8

Digital signatures with key extraction from polynomial commitments

This chapter introduces the concept of *Digital Signatures with Key Extraction (DSKE)*, and presents a DSKE construction from polynomial commitments. The work of Alpos *et al.* [10], which is a superset of this section, additionally presents a forward-forgeable signature construction, GroupForge. The construction combines a DSKE scheme with a Merkle tree and timestamps to obtain a deniable signature scheme with a fixed public key. GroupForge can replace Keyforge in the non-attributable email protocol of Specter, Park, and Green [153], hence achieving deniability without the need to continuously disclose outdated private keys. Moreover, Alpos *et al.* [10] present a second DSKE construction from hash-based digital signatures [121].

8.1 Introduction

Digital signature schemes [83] play an important role in protecting the integrity of data transmitted over the Internet. In some jurisdictions [119, 96], a digital signature applied to data can serve as evidence of the sender's authorship of the data. Moreover, the signature of a message remains valid until either the underlying signature scheme is

broken or the private key is compromised. However, as pointed out by Borisov, Goldberg, and Brewer in their work on off-the-record (OTR) communication work [28], this "long-lived" property is unsuitable for certain types of messages. For instance, if Alice wishes to communicate privately with Bob, she can encrypt her messages using Bob's public key (i.e., for confidentiality) and sign them with her private key (i.e., for authenticity). However, if Eve compromises Bob's computer at some point in the future, Eve will be able to read all of Bob's previous messages from Alice, and use the signatures to prove to Judy that the messages indeed originated from Alice.

To address this problem, OTR messaging requires an interactive key agreement protocol between the sender and the recipient to agree on session keys before exchanging messages. However, this pair-wise key agreement required in OTR is not scalable for applications such as email protocols, in which there is often no prior end-to-end interaction among the involved parties.

Another way to achieve deniability is to simply require the sender to periodically rotate keys and publish their old private keys [84]. This method enables any party to forge signatures using the published private keys and thus offers deniability to old transcripts. In fact, this method is being suggested to offer deniability in *domain keys identified mail* (DKIM) [4], where SMTP servers sign outgoing emails on behalf of the whole domain using a single key, as a way to safeguard against email spoofing.

In this chapter we design a signature scheme that allows the recipients to verify the validity of the signature and enables the sender to gain plausible deniability. It is worth noting that the question itself is, seemingly, a contradiction due to the non-repudiation property of digital signature schemes. This work circumvents the contradiction by presenting DSKE, a signature scheme based on polynomial commitments [98] can effectively convince the recipients of the authenticity of the message for a fixed period of time, while offering the signer plausible deniability. This is achieved by introducing the notions of an *extractable set*, a set of signatures from which the private key can be extracted, and a *deniable group*, a group of recipients that can, using an extractable set of signatures, collectively reconstruct the private key.

Contributions. Our contributions are summarized as follows:

- We formally define the notion of digital signature scheme with key

extraction (DSKE). In DSKE, the signer always has an *extractable set*, a set of signatures that can be used to extract the private key.

- We construct $\mathrm{DSKE}_{\mathsf{poly}}$, a digital signature scheme from a polynomial commitment primitive. Our $\mathrm{DSKE}_{\mathsf{poly}}$ results in short signatures and allows the signers to choose the size of the extractable set.

Organization. This chapter is organized as follows. Section 8.2 presents the related work, and Section 8.3 outlines the necessary background and building blocks for DSKE. In Section 8.4 we present the formal definition of DSKE, and in Section 8.5 we provide a concrete construction of DSKE from polynomial commitment schemes. Section 8.6 concludes the chapter.

8.2 Related work

Digital signature schemes with key extraction have already been explored in the context of double authentication preventing signatures (DAPS). This primitive enables the extraction of the private key if a signer creates multiple signatures on the same content. Since DAPS are genuinely designed with the purpose of double or multiple authentication prevention [133, 19, 63], they aim at messages of special form, namely $m = (a, p)$, where a is an address and p is a payload. In case a signer signs two or more messages with the same address but different payloads, then its private key is leaked. A downside of many DAPS schemes is their limited address space, i.e., an exponentially large address space is not supported. Moreover, some schemes result in considerably larger key and signature size, compared to standard signature schemes, as they build on trapdoor identification schemes [19] or involve encryption and secret sharing [63]. DSKE, on the other hand, essentially provides key extractability as an inherent feature without making assumption on the type of message, thereby increasing its applicability. In particular, the key-extraction property in $\mathrm{DSKE}_{\mathsf{poly}}$ directly comes from the polynomial interpolation theorem.

Specter, Park, and Green formally define the notion of Forward-Forgeable Signature (FFS) [153] and show how FFS can be used to achieve deniability in the email protocol. The main idea of their scheme is to make the signatures forgeable after a fixed delay. They present two concrete constructions: KeyForge and TimeForge. In KeyForge the

email server needs to periodically publish expired keys, and in TimeForge the signers relies on a trusted, publicly verifiable time-keeping service as a source of a verifiable clock. Forgeability is then derived from the possibility of obtaining a valid proof by querying the time-keeping service after a fixed delay.

Arun, Bonneau, and Clark [13] propose a similar notion, called short-lived signatures. They use a verifiable delay function [25] and a trusted random beacon [138]. The main idea is use a disjunctive statement of the form *I know the witness (e.g., private key) OR someone solved a VDF on a specific beacon value*, hence satisfying deniability because anyone can, after some specific time, produce a valid proof by evaluating the VDF. Our work, on the other hand, offers a simpler approach without requiring costly VDF evaluations and a trusted random beacon.

8.3 Background

Notation. We express by (pk, sk) a pair of public and private keys. Moreover, we require that pk can always be efficiently derived from sk, and we denote $\mathsf{extractPK}(sk) = pk$ to be the deterministic function for doing so. For a field $\mathbb{F}$, we denote $\mathbb{F}^{\leq d}(X)$ the set of polynomials in $\mathbb{F}[X]$ with degree at most d. We denote by $\mathcal{M}$ the message space and $\mathcal{S}$ the signature space.

Hash functions. Our constructions employ the following standard properties of cryptographic hash functions. We use $H : \mathcal{K} \times \mathcal{M} \to \{0,1\}^\lambda$ to denote a family of hash functions that is parameterized by a key $k \in \mathcal{K}$ and message $m \in \mathcal{M}$ and outputs a binary string of length λ.

Definition 19 (Collision resistance [143]). A family H of hash functions is collision-resistant, if for any PPT adversary $\mathcal{A}$, the adversary's advantage in finding collisions is:

$$\Pr\left[\begin{matrix} k \xleftarrow{\$} \mathcal{K} \\ (x, x') \leftarrow \mathcal{A}(k) \end{matrix} : (x \neq x') \wedge (H(k,x) = H(k,x'))\right] \leq \mathsf{negl}(\lambda)$$

In practice, the key for standard hash functions is public; therefore, from this point, we refer to the cryptographic hash function h sampled from a family of hash functions as a fixed function $h : \mathcal{M} \to \{0,1\}^\lambda$.

Polynomial commitment schemes. A polynomial commitment scheme (PCS) allows a prover to commit to a polynomial $f(X) \in \mathbb{F}^{\leq d}(X)$ and later open $f(X)$ at arbitrary points x, revealing only the value $f(x)$. We assume a succinct PCS, where the commitment C_f to $f(X)$ and the opening proofs π consist of single group elements, for some group G. A PCS consists of the following algorithms.

- $\mathsf{Setup}(1^\lambda, d) \to (ck, vk)$: The generation algorithm takes as input a security parameter λ and a maximum degree number $d \in \mathbb{N}$ and outputs the public commitment key ck, which allows committing to polynomials in $\mathbb{F}^{\leq d}(X)$, and the public verification key vk.
- $\mathsf{Com}(ck, f(X)) \to C_f$: The commitment algorithm takes as input the commitment key ck and a polynomial $f(X) \in \mathbb{F}^{\leq d}(X)$ and outputs a commitment $C_f \in G$ to the polynomial $f(X)$.
- $\mathsf{Open}(ck, C_f, x, f(X)) \to (\pi, y)$: The opening algorithm takes as input a commitment key ck, a commitment C_f, an evaluation point x, and the polynomial $f(X)$, and outputs $y = f(x) \in \mathbb{F}$ and a proof $\pi \in G$.
- $\mathsf{Check}(vk, C_f, x, y, \pi) \to b \in \{0, 1\}$: The checking algorithm takes as input the verification key vk, the commitment C_f, a point x, the claimed evaluation y, and the opening proof π, and outputs 1 iff $y = f(x)$.

Our schemes demand the following *correctness*, *hiding*, *evaluation binding*, and *polynomial binding* properties from the polynomial commitment scheme.

Definition 20 (Correctness [98]). Let $(ck, vk) \leftarrow \mathsf{Setup}(1^\lambda, k)$, $f(X) \in \mathbb{F}^{\leq d}(X)$, and $C_f \leftarrow \mathsf{Com}(ck, f(X))$. Then for any (π, y) output by $\mathsf{Open}(ck, C_f, x, f(X))$, we have that $\mathsf{Check}(vk, C_f, x, y, \pi) \to 1$.

Definition 21 (Computational hiding [98]). Given (ck, vk), the commitment C_f, and up to d valid openings (y_i, π_i) for points x_i, where $i \in \{1, \ldots, d\}$, no PPT adversary can determine the value $f(x')$, for $x' \notin \{x_1, \ldots, x_d\}$, except with negligible probability.

Definition 22 (Evaluation binding [98]). Given (ck, vk), no PPT adversary can compute commitment C_f, point x, and two openings (π_1, y_1), (π_2, y_2) for x, such that $\mathsf{Check}(vk, C, x, y_1, \pi_1) = 1$, $\mathsf{Check}(vk, C, x, y_2, \pi_2) = 1$, and $y_1 \neq y_2$.

Definition 23 (Polynomial binding [98]). Given (ck, vk), no PPT

adversary can compute polynomials $f(X)$ and $f'(X)$, such that $f(X) \neq f'(X)$ and $\mathsf{Com}(ck, f(X)) = \mathsf{Com}(ck, f'(X))$.

KZG polynomial commitment scheme. We now present a concrete polynomial commitment construction, the KZG [98] scheme. It works over a bilinear pairing group $\mathbb{G} = \langle e, G, G_t \rangle$, where G is a group of prime order p, e is a symmetric pairing $e : G \times G \to G_t$, g is a generator of G, and $h \in G$.

- $\mathsf{Setup}\ (1^\lambda, d) \to (\mathbb{G}, \mathsf{ck}, \mathsf{vk})$: the algorithm outputs a representation of the bilinear group $\mathbb{G}$, commitment key $ck = \{g, g^\alpha, \ldots, g^{\alpha^d}\}$, and verification key $vk = h^\alpha$, for an $\alpha \in \mathbb{Z}_p$ that is destroyed after setup.
- $\mathsf{Com}(ck, f(X)) \to C_f$: the algorithm computes $C_f = g^{f(\alpha)}$ using ck and outputs C_f.
- $\mathsf{Open}(ck, C_f, x, f(X)) \to (\pi, y)$: the algorithm computes $y = f(x)$ and the quotient polynomial $q(X) = \frac{f(X)-y}{X-x}$, and outputs y and $\pi = C_q = \mathsf{Com}(ck, q(X))$.
- $\mathsf{Check}(vk, C_f, x, y, \pi) \to b \in \{0, 1\}$: the algorithm outputs 1 if $e(C_f \cdot g^{-y}, h) = e(C_q, h^\alpha \cdot h^{-x})$, and 0 otherwise.

The KZG scheme satisfies the *correctness*, *computational hiding*, *evaluation binding*, and *polynomial binding* properties [98].

8.4 Digital signatures with key extraction (DSKE)

In this section, we formally define the notion of Digital Signatures with Key Extraction (DSKE). We adopt the standard digital signature definition and introduce a new algorithm to capture the capability of extracting the private key from a set of signatures.

Definition 24 ((k, δ)-Digital signature with key extraction). A signature scheme, Σ, with key extraction consists of five algorithms:

- $\mathsf{Setup}(1^\lambda) \to par$: The setup algorithm takes a security parameter 1^λ and outputs a set of public parameters, par. This algorithm runs once, and the public parameters are implicitly input to all subsequent algorithms.

- $\mathsf{KeyGen}() \to (pk, sk)$: The *probabilistic* generation algorithm outputs a pair (pk, sk) of public key and private key.
- $\mathsf{Sign}(sk, m) \to \sigma$: The signing algorithm is a *probabilistic* algorithm that takes a private key sk and a message m from the message space $\mathcal{M}$ as input and outputs a signature σ in the signature space $\mathcal{S}$.
- $\mathsf{Verify}(pk, m, \sigma) \to b \in \{0, 1\}$: The verifying algorithm is a *deterministic* algorithm that takes a public key pk, a message m, and a signature σ, and outputs the validity of the signature, $b \in \{0, 1\}$.
- $\mathsf{Extract}(\{(m_i, \sigma_i)\}_{i\in[k]}, pk) \to sk$: The extraction algorithm is a *probabilistic* algorithm that takes as input a set of distinct message-signature pairs $\{m_i, \sigma_i\}_{i\in[k]}$, such that $\sigma_i \leftarrow \mathsf{Sign}(sk, m_i)$, and the public key pk, and outputs the underlying private key sk with probability δ and $\perp$ with probability $1 - \delta$.

Apart from the straightforward correctness definition, we consider two other properties of DSKE: unforgeability and the existence of an extractable set. The security of digital signature is defined through the following experiment.

The d-times signature experiment $\mathsf{SignExp}^d_{\mathcal{A},\Sigma}(\lambda)$.

1. $\mathsf{Setup}(1^\lambda)$ and $\mathsf{KeyGen}()$ are run to obtain keys (pk, sk).
2. $\mathcal{A}$ is given pk and can ask up to d queries to the signing oracle $\mathsf{Sign}(sk, \cdot)$. Let $Q_{\mathcal{A}}^{\mathsf{Sign}(sk,\cdot)} = \{m_i\}_{i\in[d]}$ be the set of all messages for which $\mathcal{A}$ queries $\mathsf{Sign}(sk, \cdot)$, where the i^{th} query is a message $m_i \in \mathcal{M}$. Eventually, $\mathcal{A}$ outputs a pair $(m^*, \sigma^*) \in \mathcal{M} \times \mathcal{S}$.
3. The output of the experiment is defined to be 1 if and only if $m^* \notin Q_{\mathcal{A}}^{\mathsf{Sign}(sk,\cdot)}$ and $\mathsf{Verify}(pk, m^*, \sigma^*) = 1$.

Definition 25 (Existential unforgeability). A digital signature scheme Σ is *existentially unforgeable under a d-times adaptive chosen-message attack*, or *d-times-secure*, if for all PPT adversaries $\mathcal{A}$ the success probability in the previous experiment is negligible, that is,

$$\Pr[\mathsf{SignExp}^d_{\mathcal{A},\Sigma}(\lambda) = 1] \leq \mathsf{negl}(\lambda).$$

Definition 26 (Extractable set). A digital signature scheme has a (k, δ)-extractable set when the extraction algorithm $\mathsf{Extract}(\cdot)$ on input k distinct message-signature pairs $\{(m_i, \sigma_i)\}_{i\in[k]}$ and the public key pk,

such that each σ_i is a valid signature on m_i under pk, outputs the private key sk with probability δ. That is,

$$\Pr\left[\begin{array}{l} m_i \in \mathcal{M},\ \text{s.t. } m_i \neq m_j,\ \text{for } i,j \in [k], i \neq j \\ (pk, sk) \leftarrow \mathsf{KeyGen}() \\ \sigma_i \leftarrow \mathsf{Sign}(sk, m_i) \\ \mathsf{Extract}\left(\{(m_i, \sigma_i)\}_{i\in[k]}, pk\right) \rightarrow sk' \end{array} : sk = sk'\right] = \delta$$

8.5 DSKE from polynomial commitment schemes

In the following, we assume a degree bound $d \in \mathbb{Z}$ and a polynomial degree $\ell \in \mathbb{Z}$ that satisfy $1 \leq \ell \leq d$. The idea is to use the polynomial $f(X)$ of degree ℓ as the private key. Then the signature on a message m is the evaluation of $f(X)$ at point $x = h(m)$, where h is a collision-resistant hash function. For key extraction we employ polynomial interpolation: any set of $\ell + 1$ valid message-signature pairs (m_i, σ_i) can reconstruct $f(X)$. $\mathrm{DSKE}_{\mathsf{poly}}$ works as follows.

- $\mathsf{Setup}(1^\lambda, d)$: On input the security parameter λ and degree $d \in \mathbb{N}$, the algorithm runs $\Pi.\mathsf{Setup}(1^\lambda, d)$ to obtain (ck, vk), which allows the signer to commit to polynomials in $\mathbb{F}^{\leq d}(X)$, and it samples a collision-resistant hash function $h : \mathcal{M} \rightarrow \mathbb{F}$. The public parameters, par, contain ck, vk, d, and the specification of h.
- $\mathsf{KeyGen}(\ell)$: On input $1 \leq \ell \leq d$, sample $f(X) \xleftarrow{\$} \mathbb{F}^\ell(X)$ as an ℓ-degree polynomial, compute $\Pi.\mathsf{Com}(ck, f(X)) \rightarrow C_f$, and set $sk = f(X)$, $pk = C_f$.
- $\mathsf{Sign}(sk, m)$: Parse $sk = f(X)$, compute $x = h(m)$, and run $\Pi.\mathsf{Open}(ck, C_f, x, f(X)) \rightarrow (\pi, y)$. Output the signature, $\sigma = (\pi, y)$.
- $\mathsf{Verify}(pk, m, \sigma)$: Parse $pk = C_f$ and $\sigma = (\pi, y)$, compute $x = h(m)$, and output $\Pi.\mathsf{Check}(vk, C_f, x, y, \pi) \in \{0, 1\}$.
- $\mathsf{Extract}(\{(m_i, \sigma_i)\}_{i\in[k]}, pk)$: If $k \leq \ell$, or if m_i are not all distinct, then return $\perp$. If $\mathsf{Verify}(pk, m_i, \sigma_i) = 0$ for some $i \in [k]$, then return $\perp$. Otherwise, compute $x_i = h(m_i)$ and parse $\sigma_i = (\pi_i, y_i)$, for $i \in [k]$. The (at least $\ell+1$) pairs (x_i, y_i) interpolate the unique polynomial $\phi(X) \in \mathbb{F}^\ell(X)$, where $\lambda_i(X)$ are the Lagrange coeffi-

cients: $\phi(X) = \sum_{i\in[k]} y_i\lambda_i(X)$ and $\lambda_i(X) = \prod_{m\in[k]}^{m\neq i} \frac{X-x_m}{x_i-x_m}$.

Remarks on the degree of $f(X)$. The extraction of the private key from $k \geq \ell + 1$ points requires the signer to commit to a polynomial of degree at most ℓ. As the publicly available information in ck allows the signer to commit to any polynomial in $\mathbb{F}^{\leq d}(X)$, stronger properties, such as *strong correctness* [98], *bounded-polynomial extractability* [116], and *knowledge soundness* [26], have been formulated in the literature to enforce the claimed degree on $f(X)$. However, the signer in our scheme is allowed to choose any $\ell \in [1, d]$ and has no incentive to commit to a polynomial of degree larger than ℓ, as that would cost them the deniability, as we discuss in the following sections. Hence, we can assume that $f(X)$ is indeed of degree ℓ and do not require Π to satisfy any stronger property.

Theorem 22 (Existential unforgeability). *Assuming the underlying polynomial commitment scheme* Π *satisfies the* computational hiding, evaluation binding, *and* polynomial binding *properties, and that h is a collision-resistant hash function, the DSKE scheme $DSKE_{\mathsf{poly}}$ is* existentially unforgeable *under an ℓ-times adaptive chosen-message attack. That is,*

$$\Pr[\mathsf{SignExp}^{\ell}_{\mathcal{A},\Sigma_{\mathsf{poly}}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$$

Proof. Let $\mathcal{A}$ be a PPT adversary breaking the signature scheme $\mathrm{DSKE}_{\mathsf{poly}}$. We construct a PPT algorithm $\mathcal{B}$ that runs $\mathcal{A}$ as a subroutine and attacks the hiding property of the polynomial commitment scheme Π, given that Π is evaluation and polynomial binding and that h is collision-resistant. Specifically, $\mathcal{B}$ receives from its challenger up to ℓ openings $\big(i_j, f(i_j), w_{i_j}\big)$, for $i_j \in \mathbb{F}$ and $j \in [\ell]$, and for $f(X)$ not known to $\mathcal{B}$. It outputs (i^*, y^*) for unqueried i^* and wins if $y^* = f(i^*)$. Algorithm $\mathcal{B}$ also receives C_f and d as input and is given access to ck, vk. Algorithm $\mathcal{B}$ works as follows:

- Initiate $\mathcal{A}$ with input $pk = C_f$, and create an empty set S_{quer}.
- Whenever $\mathcal{A}$ requests a signature on message m, compute $x = h(m)$ and check whether $x \in S_{\mathrm{quer}}$. If this is the case, then $\mathcal{B}$ has already asked his challenger for the opening of point x, so $\mathcal{B}$ does not have to ask again. Otherwise, add x to S_{quer} and obtain the opening (π, y) of point x from the challenger of $\mathcal{B}$. Return $\sigma = (\pi, y)$ to $\mathcal{A}$.

- If $\mathcal{A}$ fails to output a valid forgery on an unqueried message, then abort. Otherwise $\mathcal{A}$ has output a message m^* and a forgery $\sigma^* = (\pi^*, y^*)$ on m^*. We assume wlog $\mathcal{A}$ has made ℓ signature queries (if not, $\mathcal{B}$ queries these values itself) and hence $\mathcal{B}$ has openings (π_i, y_i) for points x_i, with $i \in [\ell]$. Calculate $x^* = h(m^*)$. If $x^* \in S_{\text{quer}}$, then set $\mathsf{bad}_1 \leftarrow 1$ and abort. Otherwise interpolate $f'(X) \in \mathbb{F}^{\ell}(X)$ from the $\ell+1$ points $\{(x_1, y_1), \ldots, (x_\ell, y_\ell), (x^*, y^*)\}$ and compute $C_{f'} = \Pi.\mathsf{Com}(ck, f'(X))$. $\mathcal{B}$ distinguishes two cases.
  1. If $C_{f'} \neq C_f$, then set $\mathsf{bad}_2 \leftarrow 1$ and abort.
  2. Otherwise, output (x^*, y^*). Observe that, even though $C_{f'} = C_f$, it could still be the case that $f(X) \neq f'(X)$.

Denote by $\Pr[\mathsf{HidExp}^{\ell}_{\mathcal{B},\Pi}(\lambda) = 1]$ the probability that $\mathcal{B}$ wins the game above, by Output the event that $\mathcal{B}$ outputs some (x^*, y^*), and by bad_3 the event that $f(X) \neq f'(X)$. Observe that $\mathcal{B}$ wins if and only if Output happens and bad_3 does not happen. Moreover, Output happens if $\mathcal{A}$ succeeds in forging a valid signature and bad_1 and bad_2 do not happen. Therefore, we have:

$$
\begin{aligned}
&\Pr[\mathsf{HidExp}^{\ell}_{\mathcal{B},\Pi}(\lambda) = 1] \\
&\quad = \Pr[\mathsf{Output} \wedge \overline{\mathsf{bad}_3}] fff \geq \Pr[\mathsf{Output}] - \Pr[\mathsf{bad}_3] \\
&\quad = \Pr[\mathsf{SignExp}^{\ell}_{\mathcal{A},\Sigma_{\mathsf{poly}}}(\lambda) = 1 \wedge \overline{\mathsf{bad}_1} \wedge \overline{\mathsf{bad}_2}] - \Pr[\mathsf{bad}_3] \\
&\quad \geq \Pr[\mathsf{SignExp}^{\ell}_{\mathcal{A},\Sigma_{\mathsf{poly}}}(\lambda) = 1] - \Pr[\mathsf{bad}_1] - \Pr[\mathsf{bad}_2] \\
&\qquad - \Pr[\mathsf{bad}_3]
\end{aligned}
$$

For the bad events, we have the following.

1. The event bad_1 implies that $\mathcal{A}$ breaks the collision resistance property of $\mathcal{A}$, which is assumed secure, hence $\Pr[\mathsf{bad}_1] = \mathsf{negl}(\lambda)$.
2. Event bad_2 implies $f'(X) \neq f(X)$, and hence it must be that (x^*, y^*) is not a point of $f(X)$, i.e., $f(x^*) \neq y^*$. Since $\mathcal{A}$ succeeded, point (x^*, y^*) and proof π^* satisfy $\Pi.\mathsf{Check}(vk, C_f, x^*, y^*, \pi^*) = 1$. But in this case, $\mathcal{B}$ can break the *evaluation binding* of Π in the following way. It asks for the opening of point x^*, hence obtaining (π, y), where $y = f(x^*)$. This destroys any hopes of $\mathcal{B}$ to break the hiding property, but can attack evaluation binding, using the points (x^*, y^*) and (x^*, y), for which $y \neq y^*$, $\Pi.\mathsf{Check}(vk, C_f, x^*, y^*, \pi^*) = 1$, and $\Pi.\mathsf{Check}(vk, C_f, x^*, y, \pi) = 1$. Since by assumption Π satisfies the evaluation-binding property,

we get $\Pr[\mathsf{bad}_2] = \mathsf{negl}(\lambda)$.

3. Event bad_3 would violate the polynomial-binding property of Π, since $f(X) \neq f'(X)$ and $\mathsf{Com}(ck, f(X)) = \mathsf{Com}(ck, f'(X))$, thus $\Pr[\mathsf{bad}_3] = \mathsf{negl}(\lambda)$.

From the above, and from the assumption that Π is a hiding PCS (i.e., $\Pr[\mathsf{HidExp}^{\ell}_{\mathcal{B},\Pi}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$), we get

$$\begin{aligned}\Pr[\mathsf{SignExp}^{\ell}_{\mathcal{A},\Sigma_{\mathsf{poly}}}(\lambda) = 1] \leq & \Pr[\mathsf{HidExp}^{\ell}_{\mathcal{B},\Pi}(\lambda) = 1] + \Pr[\mathsf{Bad}_1] + \\ & \Pr[\mathsf{Bad}_2] + \Pr[\mathsf{Bad}_3] \\ \leq & \mathsf{negl}(\lambda)\end{aligned}$$

□

Theorem 23 (Extractable set). *Assuming the underlying polynomial commitment scheme* Π *satisfies the* evaluation binding *property, the DSKE scheme* $DSKE_{\mathsf{poly}}$ *has a* $(k, 1 - \mathsf{negl}(\lambda))$*-extractable set for any* $k \geq \ell + 1$*. That is,*

$$\begin{aligned}\delta &= \Pr\left[\begin{array}{l} m_i \in \mathcal{M},\ \textit{for } i \in [k] \\ m_i \neq m_j,\ \textit{for } i, j \in [k], i \neq j,\ \textit{and } k \geq \ell + 1 \\ \sigma_i \leftarrow \mathsf{Sign}(sk, m_i),\ \textit{for } i \in [k] \\ \mathsf{Extract}\left(\{(m_i, \sigma_i)\}_{i \in [k]}, pk\right) \rightarrow sk' \end{array} : sk = sk'\right] \\ &= 1 - \mathsf{negl}(\lambda)\end{aligned}$$

Proof. The proof follows from two facts. First, by assuming that the signer does not commit to polynomials of degree bigger than ℓ. Second, from the *evaluation binding property*, and since the points (x_i, y_i) correspond to valid signatures, we know that $y_i = f(x_i)$, for some polynomial $f(X) \in \mathbb{F}^{\leq \ell}(X)$ and for all $i \in [k]$, except with negligible probability. Due to the uniqueness of polynomial interpolation, we know that any $\ell + 1$ distinct points (x_i, y_i) define a unique polynomial $\phi(X)$ of degree at most ℓ, hence $\phi(X)$ must be the same as $f(X)$, hence $sk = sk'$ with probability $1 - \mathsf{negl}(\lambda)$. □

8.6 Discussion

Conclusion. In this chapter we have defined the concept of a signature scheme with key extraction. We present a concrete construction

based on polynomial commitment schemes, and a formal proof of security, demonstrating that signers can consistently achieve deniability by presenting a set of signatures, which in turn can be used to regenerate old private keys.

Applications. The full version [10] of this work proposes GroupForge, which combines DSKE with Merkle hash trees and timestamps to provide properties akin to KeyForge [153]. Hence, GroupForge has potential applications in non-attributable email protocols, eliminating the need for email servers to continuously publish old key material. Moreover, DSKE can find applications in scenarios where the need for authenticity is momentary, and deniability is desired in the long term, such as electronic voting and monetary donations.

Chapter 9

Conclusion

In this thesis we presented distributed protocols with threshold and general trust assumptions.

The first part concerned distributed protocols. We started by specifying, encoding, and deploying general trust assumptions in the HotStuff consensus protocol. Our benchmarks suggest that general trust assumptions can be efficiently supported in consensus. We then explored how different systems, each with its own threshold or general trust assumptions, defined on disjoint or intersecting sets of parties, can be composed into a single unified system. This enables the parties running a distributed protocol, such us consensus, or the nodes of a blockchain, to dynamically, deterministically, and non-interactively merge and work together. Last but not least, we extended the results of Guerraoui *et al.* [86] to the ERC20 smart contract, one of the most widely adopted smart contracts on Ethereum, and proved that synchronization is only required for certain well-defined sets of parties. This result can potentially catalyze the development of more efficient and scalable distributed and decentralized systems.

The second part of this thesis focused on distributed cryptographic schemes. Returning to the topic of general trust assumptions, we first described, proved, and benchmark three distributed cryptographic schemes, namely a verifiable secret sharing, a common coin, and a distributed signature scheme. Our results suggest that general assumptions can provide enhanced resilience and expanded expressibility at no or insignificant extra cost. Following this, we proposed a concretely efficient asynchronous common-coin protocol, inherently

supporting proof-of-stake and dynamic participation. This approach could replace the existing randomness-generation mechanisms in prevalent blockchains like Algorand and Cardano, which currently rely on timing assumptions. Finally, we presented a digital signature scheme that allows the recipients to verify the validity of the signature, while enabling the sender to gain plausible deniability. Unlike the state-of-the-art protocol by Specter, Park, and Green [153], our scheme does not require the constant publication of old private keys or rotation of public keys.

We anticipate that our findings can significantly contribute to the advancement of distributed systems, blockchains, and distributed cryptographic schemes across several aspects, such as resilience to failures, the expressiveness and freedom of trust assumptions they provide to users, scalability, efficiency, and deniability.

Bibliography

[1] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, and G. Stern, "Bingo: Adaptively secure packed asynchronous verifiable secret sharing and asynchronous distributed key generation," *IACR Cryptol. ePrint Arch.*, p. 1759, 2022.

[2] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Reaching consensus for asynchronous distributed key generation," in *PODC*, pp. 363–373, ACM, 2021.

[3] S. Agrawal, P. Mohassel, P. Mukherjee, and P. Rindal, "Dise: Distributed symmetric-key encryption," in *CCS*, pp. 1993–2010, ACM, 2018.

[4] E. Allman, J. Callas, M. Delany, M. Libbey, J. Fenton, and M. Thomas, "Domainkeys identified mail (DKIM) signatures." https://doi.org/10.17487/RFC5617, 2007.

[5] O. Alpos and C. Cachin, "Consensus beyond thresholds: Generalized byzantine quorums made live," in *SRDS*, pp. 21–30, IEEE, 2020.

[6] O. Alpos and C. Cachin, "Do not trust in numbers: Practical distributed cryptography with general trust," in *SSS*, vol. 14310 of *Lecture Notes in Computer Science*, pp. 536–551, Springer, 2023.

[7] O. Alpos, C. Cachin, S. H. Kamp, and J. B. Nielsen, "Practical large-scale proof-of-stake asynchronous total-order broadcast," in *AFT*, vol. 282 of *LIPIcs*, pp. 31:1–31:22, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[8] O. Alpos, C. Cachin, G. A. Marson, and L. Zanolini, "On the synchronization power of token smart contracts," in *ICDCS*, pp. 640–651, IEEE, 2021.

[9] O. Alpos, C. Cachin, and L. Zanolini, "How to trust strangers: Composition of byzantine quorum systems," in *SRDS*, pp. 120–131, IEEE, 2021.

[10] O. Alpos, Z. Wang, A. Kavousi, S. Y. Chau, D. Le, and C. Cachin, "DSKE: digital signature with key extraction," *IACR Cryptol. ePrint Arch.*, p. 1753, 2022.

[11] L. Alvisi, E. T. Pierce, D. Malkhi, M. K. Reiter, and R. N. Wright, "Dynamic byzantine quorum systems," in *DSN*, pp. 283–292, IEEE Computer Society, 2000.

[12] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek, "Secure multiparty computations on bitcoin," *Commun. ACM*, vol. 59, no. 4, pp. 76–84, 2016.

[13] A. Arun, J. Bonneau, and J. Clark, "Short-lived zero-knowledge proofs and signatures," in *ASIACRYPT (3)*, vol. 13793 of *Lecture Notes in Computer Science*, pp. 487–516, Springer, 2022.

[14] P. Aublin, R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic, "The next 700 BFT protocols," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 12:1–12:45, 2015.

[15] L. Babai, A. Gál, and A. Wigderson, "Superpolynomial lower bounds for monotone span programs," *Combinatorica*, vol. 19, no. 3, pp. 301–319, 1999.

[16] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling blockchain innovations with pegged sidechains." https://blockstream.com/sidechains.pdf.

[17] P. S. L. M. Barreto, B. Lynn, and M. Scott, "Constructing elliptic curves with prescribed embedding degrees," in *SCN*, vol. 2576 of *Lecture Notes in Computer Science*, pp. 257–267, Springer, 2002.

[18] A. Beimel, *Secure Schemes for Secret Sharing and Key Distribution*. PhD thesis, Technion, 1996.

[19] M. Bellare, B. Poettering, and D. Stebila, "Deterring certificate subversion: Efficient double-authentication-preventing signatures," in *Public Key Cryptography (2)*, vol. 10175 of *Lecture Notes in Computer Science*, pp. 121–151, Springer, 2017.

[20] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract)," in *PODC*, pp. 27–30, ACM, 1983.

[21] J. C. Benaloh and J. Leichter, "Generalized secret sharing and monotone functions," in *CRYPTO*, vol. 403 of *Lecture Notes in Computer Science*, pp. 27–35, Springer, 1988.

[22] F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin, "Can a public blockchain keep a secret?," in *TCC (1)*, vol. 12550 of *Lecture Notes in Computer Science*, pp. 260–290, Springer, 2020.

[23] E. Blum, J. Katz, C. Liu-Zhang, and J. Loss, "Asynchronous byzantine agreement with subquadratic communication," in *TCC (1)*, vol. 12550 of *Lecture Notes in Computer Science*, pp. 353–380, Springer, 2020.

[24] A. Boldyreva, "Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme," in *Public Key Cryptography*, vol. 2567 of *Lecture Notes in Computer Science*, pp. 31–46, Springer, 2003.

[25] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch, "Verifiable delay functions," in *CRYPTO (1)*, vol. 10991 of *Lecture Notes in Computer Science*, pp. 757–788, Springer, 2018.

[26] D. Boneh, J. Drake, B. Fisch, and A. Gabizon, "Halo infinite: Proof-carrying data from additive polynomial commitments," in *CRYPTO (1)*, vol. 12825 of *Lecture Notes in Computer Science*, pp. 649–680, Springer, 2021.

[27] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the weil pairing," *J. Cryptol.*, vol. 17, no. 4, pp. 297–319, 2004.

[28] N. Borisov, I. Goldberg, and E. A. Brewer, "Off-the-record communication, or, why not to use PGP," in *WPES*, pp. 77–84, ACM, 2004.

[29] E. F. Brickell, "Some ideal secret sharing schemes," in *EUROCRYPT*, vol. 434 of *Lecture Notes in Computer Science*, pp. 468–475, Springer, 1989.

[30] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018.

[31] C. Cachin, "Distributing trust on the internet," in *DSN*, pp. 183–192, IEEE Computer Society, 2001.

[32] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[33] C. Cachin, B. Junker, and A. Sorniotti, "On limitations of using cloud storage for data replication," in *DSN Workshops*, pp. 1–6, IEEE Computer Society, 2012.

[34] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *CRYPTO*, vol. 2139 of *Lecture Notes in Computer Science*, pp. 524–541, Springer, 2001.

[35] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *J. Cryptol.*, vol. 18, no. 3, pp. 219–246, 2005.

[36] C. Cachin, S. Schubert, and M. Vukolic, "Non-determinism in byzantine fault-tolerant replication," in *OPODIS*, vol. 70 of *LIPIcs*, pp. 24:1–24:16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[37] C. Cachin and B. Tackmann, "Asymmetric distributed trust," in *OPODIS*, vol. 153 of *LIPIcs*, pp. 7:1–7:16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[38] C. Cachin and L. Zanolini, "Asymmetric asynchronous byzantine consensus," in *DPM/CBT@ESORICS*, vol. 13140 of *Lecture Notes in Computer Science*, pp. 192–207, Springer, 2021.

[39] J. Camenisch, M. Drijvers, T. Hanke, Y. Pignolet, V. Shoup, and D. Williams, "Internet computer consensus," in *PODC*, pp. 81–91, ACM, 2022.

[40] R. Canetti and T. Rabin, "Fast asynchronous byzantine agreement with optimal resilience," in *STOC*, pp. 42–51, ACM, 1993.

[41] I. Cascudo and B. David, "SCRAPE: scalable randomness attested by public entities," in *ACNS*, vol. 10355 of *Lecture Notes in Computer Science*, pp. 537–556, Springer, 2017.

[42] I. Cascudo and B. David, "ALBATROSS: publicly attestable batched randomness based on secret sharing," in *ASIACRYPT (3)*, vol. 12493 of *Lecture Notes in Computer Science*, pp. 311–341, Springer, 2020.

[43] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[44] D. Chaum and T. P. Pedersen, "Wallet databases with observers," in *CRYPTO*, vol. 740 of *Lecture Notes in Computer Science*, pp. 89–105, Springer, 1992.

[45] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos, "ALGORAND AGREEMENT: super fast and partition resilient byzantine agreement," *IACR Cryptol. ePrint Arch.*, p. 377, 2018.

[46] K. Choi, A. Arun, N. Tyagi, and J. Bonneau, "Bicorn: An optimistically efficient distributed randomness beacon," *IACR Cryptol. ePrint Arch.*, p. 221, 2023.

[47] K. Choi, A. Manoj, and J. Bonneau, "Sok: Distributed randomness beacons," *IACR Cryptol. ePrint Arch.*, p. 728, 2023.

[48] A. Choudhury, "Almost-surely terminating asynchronous byzantine agreement against general adversaries with optimal resilience," in *ICDCN*, pp. 167–176, ACM, 2023.

[49] S. Cohen, I. Keidar, and A. Spiegelman, "Not a coincidence: Sub-quadratic asynchronous byzantine agreement WHP," in *DISC*, vol. 179 of *LIPIcs*, pp. 25:1–25:17, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[50] D. Collins, R. Guerraoui, J. Komatovic, P. Kuznetsov, M. Monti, M. Pavlovic, Y. Pignolet, D. Seredinschi, A. Tonkikh, and A. Xygkis, "Online payments by merely broadcasting messages," in *DSN*, pp. 26–38, IEEE, 2020.

[51] R. Cramer, I. Damgård, and U. M. Maurer, "General secure multi-party computation from any linear secret-sharing scheme," in *EUROCRYPT*, vol. 1807 of *Lecture Notes in Computer Science*, pp. 316–334, Springer, 2000.

[52] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. E. Kosba, A. Miller, P. Saxena, E. Shi, E. G. Sirer, D. Song, and R. Wattenhofer, "On scaling decentralized blockchains - (A position paper)," in *Financial Cryptography Workshops*, vol. 9604 of *Lecture Notes in Computer Science*, pp. 106–125, Springer, 2016.

[53] Crypto Forum Research Group (CFRG), Internet Research Task Force (IRTF), "Pairing-Friendly Curves." https://www.ietf.org/archive/id/draft-irtf-cfrg-pairing-friendly-curves-10.html, 2021.

[54] J. Dafflon, J. Baylina, and T. Shababi, "EIP-777: ERC777 Token Standard." https://eips.ethereum.org/EIPS/eip-777.

[55] I. Damgård, Y. Desmedt, M. Fitzi, and J. B. Nielsen, "Secure protocols with asymmetric trust," in *ASIACRYPT*, vol. 4833 of *Lecture Notes in Computer Science*, pp. 357–375, Springer, 2007.

[56] S. Das, V. Krishnan, I. M. Isaac, and L. Ren, "Spurt: Scalable distributed randomness beacon with transparent setup," in *IEEE Symposium on Security and Privacy*, pp. 2502–2517, IEEE, 2022.

[57] S. Das, T. Yurek, Z. Xiang, A. Miller, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous distributed key generation," in *IEEE Symposium on Security and Privacy*, pp. 2518–2534, IEEE, 2022.

[58] Data Sharing Coalition, "Developing a safe and trusted collaboration environment to monitor and combat human trafficking," 2021. https://datasharingcoalition.eu/2021/developing-a-safe-and-trusted-collaboration-environment-to-monitor-and-combat-human-trafficking.

[59] B. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *EUROCRYPT (2)*, vol. 10821 of *Lecture Notes in Computer Science*, pp. 66–98, Springer, 2018.

[60] B. David, B. Magri, C. Matt, J. B. Nielsen, and D. Tschudi, "Gearbox: Optimal-size shard committees by leveraging the safety-liveness dichotomy," in *CCS*, pp. 683–696, ACM, 2022.

[61] V. Daza, J. Herranz, and G. Sáez, "Constructing general dynamic group key distribution schemes with decentralized user join," in *ACISP*, vol. 2727 of *Lecture Notes in Computer Science*, pp. 464–475, Springer, 2003.

[62] V. Daza, J. Herranz, and G. Sáez, "On the computational security of a distributed key distribution scheme," *IEEE Trans. Computers*, vol. 57, no. 8, pp. 1087–1097, 2008.

[63] D. Derler, S. Ramacher, and D. Slamanig, "Short double- and n-times-authentication-preventing signatures from ECDSA and more," in *EuroS&P*, pp. 273–287, IEEE, 2018.

[64] Y. Desmedt, "Society and group oriented cryptography: A new concept," in *CRYPTO*, vol. 293 of *Lecture Notes in Computer Science*, pp. 120–127, Springer, 1987.

[65] Drand, "A distributed randomness beacon daemon," 2022. https://drand.love.

[66] C. Dwork, N. A. Lynch, and L. J. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[67] S. Dziembowski, L. Eckey, S. Faust, and D. Malinowski, "Perun: Virtual payment hubs over cryptocurrencies," in *IEEE Symposium on Security and Privacy*, pp. 106–123, IEEE, 2019.

[68] ECRYPT-CSA, "Algorithms, key size and protocols report," *H2020-ICT-2014 – Project 645421*, 2018. https://www.ecrypt.eu.org/csa/documents/D5.4-FinalAlgKeySizeProt.pdf.

[69] W. Entriken, D. Shirley, J. Evans, and N. Sachs, "EIP-721: ERC-721 Non-Fungible Token Standard." https://eips.ethereum.org/EIPS/eip-721, 2018.

[70] R. Eriguchi and K. Nuida, "Homomorphic secret sharing for multipartite and general adversary structures supporting parallel evaluation of low-degree polynomials," in *ASIACRYPT (2)*, vol. 13091

of *Lecture Notes in Computer Science*, pp. 191–221, Springer, 2021.

[71] Ethereum Foundation, "Ethereum." https://ethereum.org/.

[72] Ethereum Foundation, "Ethereum Request for Comments." https://eips.ethereum.org/erc.

[73] Ethereum Foundation, "Shard chains." https://ethereum.org/en/eth2/shard-chains/.

[74] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," in *FOCS*, pp. 427–437, IEEE Computer Society, 1987.

[75] A. Fiat and A. Shamir, "How to prove yourself: Practical solutions to identification and signature problems," in *CRYPTO*, vol. 263 of *Lecture Notes in Computer Science*, pp. 186–194, Springer, 1986.

[76] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[77] D. Frey, M. Gestin, and M. Raynal, "The synchronization power (consensus number) of access-control objects: The case of allowlist and denylist," *CoRR*, vol. abs/2302.06344, 2023.

[78] J. A. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *EUROCRYPT (2)*, vol. 9057 of *Lecture Notes in Computer Science*, pp. 281–310, Springer, 2015.

[79] R. Gennaro, *Theory and practice of verifiable secret sharing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[80] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust threshold DSS signatures," in *EUROCRYPT*, vol. 1070 of *Lecture Notes in Computer Science*, pp. 354–371, Springer, 1996.

[81] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," *J. Cryptol.*, vol. 20, no. 1, pp. 51–83, 2007.

[82] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *SOSP*, pp. 51–68, ACM, 2017.

[83] S. Goldwasser, S. Micali, and R. L. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM J. Comput.*, vol. 17, no. 2, pp. 281–308, 1988.

[84] M. Green, "Ok Google: please publish your DKIM secret keys." https://blog.cryptographyengineering.com/2020/11/16/ok-google-please-publish-your-dkim-secret-keys, 2020.

[85] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi, "Scalable byzantine reliable broadcast," in *DISC*, vol. 146 of *LIPIcs*, pp. 22:1–22:16, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[86] R. Guerraoui, P. Kuznetsov, M. Monti, M. Pavlovic, and D. Seredinschi, "The consensus number of a cryptocurrency," *Distributed Comput.*, vol. 35, no. 1, pp. 1–15, 2022.

[87] S. Gupta, "A non-consensus based decentralized financial transaction processing model with support for efficient auditing." Master Thesis, Arizona State University, USA, June 2016.

[88] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[89] J. Herranz, C. Padró, and G. Sáez, "Distributed RSA signature schemes for general access structures," in *ISC*, vol. 2851 of *Lecture Notes in Computer Science*, pp. 122–136, Springer, 2003.

[90] J. Herranz and G. Sáez, "Verifiable secret sharing for general access structures, with application to fully distributed proxy signatures," in *Financial Cryptography*, vol. 2742 of *Lecture Notes in Computer Science*, pp. 286–302, Springer, 2003.

[91] M. Hirt and U. M. Maurer, "Player simulation and general adversary structures in perfect multiparty computation," *J. Cryptology*, vol. 13, no. 1, pp. 31–60, 2000.

[92] H. Howard, A. Charapko, and R. Mortier, "Fast flexible paxos: Relaxing quorum intersection for fast paxos," in *ICDCN*, pp. 186–190, ACM, 2021.

[93] IOTA Foundation, "Iota." https://www.iota.org/.

[94] M. Ito, A. Saito, and T. Nishizeki, "Secret sharing scheme realizing general access structure," *Electronics and Communications in Japan*, vol. 72, pp. 56–64, 1989.

[95] F. P. Junqueira, K. Marzullo, M. Herlihy, and L. D. Penso, "Threshold protocols in survivor set systems," *Distributed Comput.*, vol. 23, no. 2, pp. 135–149, 2010.

[96] N. Karanikolas, "Digital signature legality in different jurisdictions: Legally binding issues." https://repository.ihu.edu.gr/xmlui/handle/11544/29366, 2019.

[97] M. Karchmer and A. Wigderson, "On span programs," in *Computational Complexity Conference*, pp. 102–111, IEEE Computer Society, 1993.

[98] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *ASIACRYPT*, vol. 6477 of *Lecture Notes in Computer Science*, pp. 177–194, Springer, 2010.

[99] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All you need is DAG," in *PODC*, pp. 165–175, ACM, 2021.

[100] V. King and J. Saia, "Byzantine agreement in expected polynomial time," *J. ACM*, vol. 63, no. 2, pp. 13:1–13:21, 2016.

[101] V. King and J. Saia, "Correction to byzantine agreement in expected polynomial time, JACM 2016," *CoRR*, vol. abs/1812.10169, 2018.

[102] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," in *IEEE Symposium on Security and Privacy*, pp. 583–598, IEEE Computer Society, 2018.

[103] K. G. Larsen and M. Simkin, "Secret sharing lower bound: Either reconstruction is hard or shares are long," in *SCN*, vol. 12238 of *Lecture Notes in Computer Science*, pp. 566–578, Springer, 2020.

[104] A. K. Lenstra and B. Wesolowski, "A random zoo: sloth, unicorn, and trx," *IACR Cryptol. ePrint Arch.*, p. 366, 2015.

[105] A. B. Lewko and B. Waters, "Decentralizing attribute-based encryption," in *EUROCRYPT*, vol. 6632 of *Lecture Notes in Computer Science*, pp. 568–588, Springer, 2011.

[106] J. Li, M. H. Au, W. Susilo, D. Xie, and K. Ren, "Attribute-based signature and its applications," in *AsiaCCS*, pp. 60–69, ACM, 2010.

[107] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic, "XFT: practical fault tolerance beyond crashes," in *OSDI*, pp. 485–500, USENIX Association, 2016.

[108] N. Lohmann, "JSON for Modern C++ version 3.7.3," 2019. https://nlohmann.github.io/json/.

[109] M. Lokhava, G. Losa, D. Mazières, G. Hoare, N. Barry, E. Gafni, J. Jove, R. Malinowsky, and J. McCaleb, "Fast and secure global payments with stellar," in *SOSP*, pp. 80–96, ACM, 2019.

[110] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "Honeybadgermpc and asynchromix: Practical asynchronous MPC and its application to anonymous communication," in *CCS*, pp. 887–903, ACM, 2019.

[111] H. K. Maji, M. Prabhakaran, and M. Rosulek, "Attribute-based signatures," in *CT-RSA*, vol. 6558 of *Lecture Notes in Computer Science*, pp. 376–392, Springer, 2011.

[112] D. Malkhi, K. Nayak, and L. Ren, "Flexible byzantine fault tolerance," in *CCS*, pp. 1041–1053, ACM, 2019.

[113] D. Malkhi and M. K. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[114] D. Malkhi, M. K. Reiter, and A. Wool, "The load and availability of byzantine quorum systems," *SIAM J. Comput.*, vol. 29, no. 6, pp. 1889–1906, 2000.

[115] D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright, "Probabilistic quorum systems," *Inf. Comput.*, vol. 170, no. 2, pp. 184–206, 2001.

[116] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn, "Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings," in *CCS*, pp. 2111–2128, ACM, 2019.

[117] K. Martin, "New secret sharing schemes from old," *J. of Comb. Math. and Combin. Comput.*, vol. 14, pp. 65–77, 1993.

[118] S. Mashhadi, M. H. Dehkordi, and N. Kiamari, "Provably secure verifiable multi-stage secret sharing scheme based on monotone span program," *IET Inf. Secur.*, vol. 11, no. 6, pp. 326–331, 2017.

[119] S. Mason, *Electronic signatures in law*. University of London Press, 2016.

[120] D. Mazières, "The Stellar consensus protocol: A federated model for Internet-level consensus." Stellar, available online, https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2016.

[121] R. C. Merkle, "A certified digital signature," in *CRYPTO*, vol. 435 of *Lecture Notes in Computer Science*, pp. 218–238, Springer, 1989.

[122] S. Micali, M. O. Rabin, and S. P. Vadhan, "Verifiable random functions," in *FOCS*, pp. 120–130, IEEE Computer Society, 1999.

[123] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of BFT protocols," in *CCS*, pp. 31–42, ACM, 2016.

[124] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System." Whitepaper, http://bitcoin.org/bitcoin.pdf, 2009.

[125] M. Naor, B. Pinkas, and O. Reingold, "Distributed pseudo-random functions and kdcs," in *EUROCRYPT*, vol. 1592 of *Lecture Notes in Computer Science*, pp. 327–346, Springer, 1999.

[126] M. Naor and A. Wool, "The load, capacity, and availability of quorum systems," *SIAM J. Comput.*, vol. 27, no. 2, pp. 423–447, 1998.

[127] V. Nikov and S. Nikova, "New monotone span programs from old," *IACR Cryptology ePrint Archive*, vol. 2004, p. 282, 2004.

[128] V. Nikov, S. Nikova, B. Preneel, and J. Vandewalle, "On distributed key distribution centers and unconditionally secure proactive verifiable secret sharing schemes based on general access structure," in *INDOCRYPT*, vol. 2551 of *Lecture Notes in Computer Science*, pp. 422–436, Springer, 2002.

[129] T. Okamoto and K. Takashima, "Decentralized attribute-based encryption and signatures," *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, vol. 103-A, no. 1, pp. 41–73, 2020.

[130] C. Padró and G. Sáez, "Secret sharing schemes with bipartite access structure," *IEEE Trans. Inf. Theory*, vol. 46, no. 7, pp. 2596–2604, 2000.

[131] A. Patra, A. Choudhury, and C. P. Rangan, "Asynchronous byzantine agreement with optimal resilience," *Distributed Comput.*, vol. 27, no. 2, pp. 111–146, 2014.

[132] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *CRYPTO*, vol. 576 of *Lecture Notes in Computer Science*, pp. 129–140, Springer, 1991.

[133] B. Poettering and D. Stebila, "Double-authentication-preventing signatures," *Int. J. Inf. Sec.*, vol. 16, no. 1, pp. 1–22, 2017.

[134] J. Poon and T. Dryja, "The bitcoin lightning network: Scalable off-chain instant payments." https://lightning.network/lightning-network-paper.pdf.

[135] Protocol Labs, "Filecoin: A decentralized storage network." https://filecoin.io/filecoin.pdf, 2017.

[136] Quarkslab SAS, "Technical assessment of herumi libraries." https://blog.quarkslab.com/resources/2020-12-17-technical-assessment-of-herumi-libraries/20-07-732-REP.pdf, 2020.

[137] M. O. Rabin, "Randomized byzantine generals," in *FOCS*, pp. 403–409, IEEE Computer Society, 1983.

[138] M. O. Rabin, "Transaction protection by beacons," *J. Comput. Syst. Sci.*, vol. 27, no. 2, pp. 256–267, 1983.

[139] M. Raikwar and D. Gligoroski, "Sok: Decentralized randomness beacon protocols," in *ACISP*, vol. 13494 of *Lecture Notes in Computer Science*, pp. 420–446, Springer, 2022.

[140] M. Raynal, *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

[141] R. L. Rivest, A. Shamir, and D. Wagner, "Time-lock puzzles and timed-release crypto." https://hdl.handle.net/1721.1/149822, 1996.

[142] R. Robere, T. Pitassi, B. Rossman, and S. A. Cook, "Exponential lower bounds for monotone span programs," in *FOCS*, pp. 406–415, IEEE Computer Society, 2016.

[143] P. Rogaway and T. Shrimpton, "Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance," in *FSE*, vol. 3017 of *Lecture Notes in Computer Science*, pp. 371–388, Springer, 2004.

[144] P. Schindler, A. Judmayer, N. Stifter, and E. R. Weippl, "Hydrand: Efficient continuous distributed randomness," in *IEEE Symposium on Security and Privacy*, pp. 73–89, IEEE, 2020.

[145] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[146] D. Schwartz, N. Youngs, and A. Britto, "The Ripple protocol consensus algorithm." Ripple Labs, available online, https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

[147] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[148] V. Shoup, "Lower bounds for discrete logarithms and related problems," in *EUROCRYPT*, vol. 1233 of *Lecture Notes in Computer Science*, pp. 256–266, Springer, 1997.

[149] V. Shoup, "Practical threshold signatures," in *EUROCRYPT*, vol. 1807 of *Lecture Notes in Computer Science*, pp. 207–220, Springer, 2000.

[150] V. Shoup, *A Computational Introduction to Number Theory and Algebra Version 2*. Cambridge University Press, 2009.

[151] V. Shoup, "Number Theory Library for C++ version 11.4.3," 2020. https://www.shoup.net/ntl.

[152] Y. Sompolinsky, S. Wyborski, and A. Zohar, "PHANTOM GHOSTDAG: a scalable generalization of nakamoto consensus: September 2, 2021," in *AFT*, pp. 57–70, ACM, 2021.

[153] M. A. Specter, S. Park, and M. Green, "Keyforge: Non-attributable email from forward-forgeable signatures," in *USENIX Security Symposium*, pp. 1755–1773, USENIX Association, 2021.

[154] T. Swanson, "Consensus-as-a-service: A brief report on the emergence of permissioned, distributed ledger systems." Available online, http://www.ofnumbers.com/wp-content/uploads/2015/04/Permissioned-distributed-ledgers.pdf, 2015.

[155] E. Syta, P. Jovanovic, E. Kokoris-Kogias, N. Gailly, L. Gasser, I. Khoffi, M. J. Fischer, and B. Ford, "Scalable bias-resistant distributed randomness," in *IEEE Symposium on Security and Privacy*, pp. 444–460, IEEE Computer Society, 2017.

[156] J. R. R. Tolkien, *The fellowship of the ring: being the first part of The Lord of the Rings*. Mariner Books/Houghton Mifflin Harcourt, 2012.

[157] A. Tomescu, R. Chen, Y. Zheng, I. Abraham, B. Pinkas, G. Golan-Gueta, and S. Devadas, "Towards scalable threshold cryptosystems," in *IEEE Symposium on Security and Privacy*, pp. 877–893, IEEE, 2020.

[158] F. Victor and B. K. Lüders, "Measuring ethereum-based ERC20 token networks," in *Financial Cryptography*, vol. 11598 of *Lecture Notes in Computer Science*, pp. 113–129, Springer, 2019.

[159] W. Vogels, "Life is not a State-Machine." https://www.allthingsdistributed.com/2006/08/life_is_not_a_statemachine.html, 2006.

[160] F. Vogelsteller and V. Buterin, "EIP-20: ERC-20 Token Standard." Available online, 2015.

[161] T. Warns, F. C. Freiling, and W. Hasselbring, "Solving consensus using structural failure models," in *SRDS*, pp. 212–224, IEEE Computer Society, 2006.

[162] Web3 Foundation, "Polkadot." https://polkadot.network/.

[163] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *PODC*, pp. 347–356, ACM, 2019.

[164] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *CCS*, pp. 931–948, ACM, 2018.

[165] Q. Zhou, H. Huang, Z. Zheng, and J. Bian, "Solutions to scalability of blockchain: A survey," *IEEE Access*, vol. 8, pp. 16440–16455, 2020.

Declaration of consent

on the basis of Article 18 of the PromR Phil.-nat. 19

Name/First Name: Alpos Orestis Charilaos

Registration Number: 19-110-535

Study program: Computer Science

Bachelor ☐ Master ☐ Dissertation ☑

Title of the thesis: Distributed Protocols with Threshold and General Trust Assumptions

Supervisor: Prof. Dr. Christian Cachin

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of September 5th, 1996 and Article 69 of the University Statute of June 7th, 2011 is authorized to revoke the doctoral degree awarded on the basis of this thesis.
For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with theses submitted by others.

Bern, 10.07.2023

Place/Date

Orestis Charilaos Alpos

Signature

Curriculum Vitae

Education

University of Bern *June 2019 – Aug 2023*

- PhD researcher in computer science, Cryptology and data security lab
- Teaching assistant in Privacy and data security (Msc), Cryptography (Msc), Discrete math (Bsc)
- Supervision of Bsc and Msc student theses in distributed computing and cryptography projects

National Technical University of Athens *Sept 2011 – July 2017*

- Electrical and computer engineering, 5-year BSc-Msc, 300 ECTS
- Thesis: Classification of network attacks using machine learning
- Grade: 8.9/10 (top 6% of 2017 graduates)

Work Experience

Research Internship @ IBM Zurich *April 2021 - July 2021*

- Research on consensus protocols (Mir and HotStuff BFT), wide-area benchmarks
- Code base in Golang, bash scipts, python

Full Stack Developer @ Greek Army *Sept 2018 – May 2019*

- Front-end application using Angular, TypeScript, HTML
- Back-end API calls and databases using .Net Framework, C#, MS SQL Server

Software Developer @ Entersoft SA *Oct 2016 – Aug 2018*

- Coding and design patterns using .Net technologies and C#
- Database development and optimization using SQL Server and Azure

Publications and pre-prints

1. *Practical Large-Scale Proof-of-Stake Asynchronous Total-Order Broadcast.* Orestis Alpos, Christian Cachin, Simon Holmgaard Kamp, Jesper Buus Nielsen. AFT, vol. 282 of LIPIcs, pp. 31:1–31:22, Schloss Dagstuhl, 2023
2. *Eating sandwiches: Modular and lightweight elimination of transaction reordering attacks.* Orestis Alpos, Ignacio Amores-Sesar, Christian Cachin, Michelle Yeo. To appear in the proc. of OPODIS 2023
3. *Do Not Trust in Numbers: Practical Distributed Cryptography With General Trust.* Orestis Alpos, Christian Cachin. SSS, vol. 14310 of Lecture Notes in Computer Science, pp. 536–551, Springer, 2023
4. *DSKE: Digital Signature with Key Extraction.* Orestis Alpos, Zhipeng Wang, Alireza Kavousi, Sze Yiu Chau, Duc Le, Christian Cachin. IACR Cryptol. ePrint Arch. 2022:1753 2022
5. *On the Synchronization Power of Token Smart Contracts.* Orestis Alpos, Christian Cachin, Giorgia Azzurra Marson, Luca Zanolini. ICDCS 2021
6. *How to Trust Strangers: Composition of Byzantine Quorum Systems.* Orestis Alpos, Christian Cachin, Luca Zanolini. SRDS 2021
7. *Consensus Beyond Thresholds: Generalized Byzantine Quorums Made Live* Orestis Alpos, Christian Cachin. SRDS 2020

Service to community

- Code developer and protocol designer for ThetaCrypt, an open-source threshold-cryptographic library in Rust, developed at the University of Bern. Thetacrypt enables modular usage (over gRPC) of threshold cryptography on any distributed ledger that supports custom applications (e.g., smart contracts).
- Supervised 3 Msc and 9 Bsc theses during my PhD studies
- Sub-reviewer: SPAA '23, DISC '22, AFT '22, Asiacrypt '21, Asiacrypt '20, IEEE-TIFS and -TDSC '20-22
- Co-orgnanizer: Theory and practice of Blockchains workshop, 2021

Invited talks

- *Bringing classical distributed protocols into the heterogeneous setting with Asymmetric BQS*
  Heterogeneous trust in distributed systems workshop, AFT, October 2023
- *Practical distributed cryptography with general trust, and applications in consensus*
  Chainlink, March 2023

- *Do not trust in numbers: Practical distributed cryptography with general trust*
  IC3 winter retreat, January 2023
- *The Future of Cryptography - Challenges and Key Technologies: Panel with experts from academia and industry*
  Panelist, organized by the Swiss Cyber Defense Campus, October 2022
- *DSKE: Digital signature with key extraction*
  Cryptography and security group, Aarhus university, research visit, September 2022

(Natural) Languages

- *Greek:* Native Speaker
- *English:* Level C2, *Michigan ECPE and Cambridge CPE Certificates,* 2008
- *German:* Level B2, *Göthe Zertifikat,* 2013
- *Spanish:* Level B2, *DELE Certificado,* 2018